TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ANTTI BLÅFIELD

MOBIILISOVELLUS MERIKONTTIEN KUNNON JA LAADUN VALVONTAAN

Diplomityö

Tarkastaja: Apulaisprofessori Sami Hyrynsalmi
Tarkastaja ja aihe hyväksytty Talouden ja rakentamisen koulutusvaradekaanin päätöksellä 24. helmikuuta 2017

TIIVISTELMÄ

ANTTI BLÅFIELD: Mobiilisovellus merikonttien kunnon ja laadun valvontaan
Tampereen teknillinen yliopisto
Diplomityö, 58 sivua
Marraskuu 2017
Johtamisen ja tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto ja tiedonhallinta
Tarkastaja: apulaisprofessori Sami Hyrynsalmi

Avainsanat: mobiilisovellus, laadunvalvonta, merikontti

Tässä diplomityössä tutkitaan tietojärjestelmän arkkitehtuurin syntymistä ketterässä ohjelmistokehityksessä. Diplomityön puitteissa Euroports Rauma Oy:lle toteutettiin merikonttien kunnon ja laadun valvontaan tarkoitettu tietojärjestelmä, jolla arkkitehtuurin syntymistä voitiin arvioida sekä vastata tämän työn tutkimustavoitteeseen. Konttien tarkastukseen ja vaurioiden dokumentoimiseen käytettävä nykyinen järjestelmä, on jokseenkin vanhentunut ja tästä syystä selvitys oli tarpeellinen. Nykyisessä järjestelmässä käytettävien käsipäätteiden korvaaminen pienemmillä ja edullisemmilla mobiililaitteilla, esimerkiksi puhelimilla, olisi työn toimeksiannon mukaan hyvä asia.

Merikonttien kunnon ja laadun valvontaan tarkoitetun Container App -sovelluksen toteutuksessa käytettiin Extreme Programming (XP) -ohjelmistokehitysmenetelmää. Työssä keskityttiin toteuttamaan sovellukseen mobiili- ja web-käyttöliittymää sen verran, että työn toimeksiantajan oli mahdollista arvioida olisiko uusi järjestelmä mahdollista toteuttaa valituilla välineillä ja teknologioilla. Sovelluksen arvionnissa yksi keskeinen asia oli se millainen mobiililaitteen käytettävyys on kenttäolosuhteissa. Konttikentällä sääolosuhteet voivat olla hyvinkin haastavat ja sen vuoksi mobiilisovelluksen käyttöliittymän pitää olla visuaalisesti selkeä ja mahdollisimman yksinkertainen käyttää.

Toteutetun Container App -sovelluksen toteutuksen edetessä pystyi selvästi havaitsemaan sovelluksen arkkitehtuurin syntymisen. Arkkitehtuuri täydentyi ja osin myös muuttui ohjelmistoprojektini kolmessa toteutusjaksossa. Aluksi toteutettavan sovelluksen arkkitehtuurista oli olemassa jonkin näköistä suunnitelmaa mutta lopulliseen muotoonsa sovelluksen arkkitehtuuri kehittyi ohjelmistoprojektini kolmessa toteutusjaksossa. Ketterässä ohjelmistoprojektissani järjestelmän rakenne muodostuu asiakkaan ja kehitysryhmän valitsemien toteutettavien ominaisuuksien mukaan, inkrementaalisesti kehitysjaksosta toiseen. Sovellusta esiteltiin Europortsilla kontin tarkastajille ja yleinen vastaanotto oli positiivista. Lopullisena arviona ja näkemyksenä on, että sovelluksessa olisi potentiaalia konttien tarkastusta helpottavaan ja nopeuttavaan ratkaisuun. Itse sovellus vaatisi kuitenkin vielä melko paljon jatkokehittelyä, jotta kaikki tarvittava toiminnallisuus saataisiin siihen toteutettua.

ABSTRACT

ANTTI BLÅFIELD: Mobile application for recording sea container quality details
Tampere University of Technology
Master of Science Thesis, 58 pages
November 2017
Master's Degree Programme in Management and Information Technology
Major: Software Engineering and Data Management
Examiner: assistant professor Sami Hyrynsalmi

Keywords: mobile application, quality control, sea container

This thesis examines the architecture of the information system in the agile development of software. In the framework of the thesis, an information system for recording sea container quality details was implemented for Euroports Rauma Oy, which could be used to evaluate the design of the architecture and to answer to the research objective of this work. The current system for checking containers and documenting damages is somewhat outdated and therefore a clarification was necessary. Replacement of hand terminals used in the current system with smaller and less expensive mobile devices, such as mobile phones, would be a good thing according to the job's mandate.

The Extreme Programming (XP) software development method was used to implement the Container App -application for recording sea container quality details. The thesis focused on implementing the application on the mobile and web interface so that it was possible for the client to evaluate whether the new system could be implemented with selected tools and technologies. One key issue in evaluating an application was the usability of a mobile device in field conditions. In a container storage area, weather conditions can be very challenging and therefore the user interface of the mobile application must be visually clear and as simple as possible.

As the implemented Container App implementation progressed, it was clearly possible to detect how the architecture of the application formed. The architecture was complemented and partly changed in the three implementation cycles of my software project. At first, there was a plan for the first application architecture, but in its final form, the architecture of the application developed in the three implementation cycles of the software project. In the agile software project, the system structure is formed by the customer and the developer's chosen features, incrementally from one development point to another. The application was presented to container inspectors in Euroports and the general reception was positive. The final assessment and the view is that the application would have potential to facilitate the inspection of containers and to speed up the inspection process. However, the application itself would still require quite a lot of further development to make all the functionality required.

ALKUSANAT

Reilu kolmen vuoden tiukka rutistus diplomi-insinööriopinnoissa päättyi mielenkiintoiseen diplomityöprojektiin. Vaikka opiskelu työn ohessa on ollut välillä hyvin rankkaa niin se on ollut myös todella mielenkiintoista ja palkitsevaa. Uskon, että saan opinnoista vielä paljon hyötyä työelämässä tulevaisuudessa.

Haluan kiittää Tampereen teknillisen yliopiston Porin yksikön henkilökuntaa hyvästä opetuksesta ja miellyttävän opiskeluilmapiirin luomisesta. TTY:n Porin yksikössä on panostettu monimuotoisiin opetusmenetelmiin jotka mahdollistavat DI-tutkinnon suorittamisen työn ohessa. Erityisen kiitoksen haluan osoittaa apulaisprofessori Sami Hyrynsalmelle, joka toimi diplomityöni tarkastajana. Sain Samilta todella hyvää ohjausta koko diplomityön tekemisen ajan.

Seuraavaksi haluan kiittää Tommi Heinilää ja Sakari Kasevaa Euroports Rauma Oy:stä jotka mahdollistivat diplomityöni tekemisen valittuun aiheeseen liittyen. Tommi ja Sakari opastivat ja neuvoivat minua kiitettävästi, jotta pystyin omaksumaan tarvittavat tiedot merikonttien tarkastukseen liittyvästä prosessista. Sain heiltä myös hyvin palautetta lopputyöni edetessä.

Suuren kiitoksen ansaitsee myös kollegani Jaakko Sirkki, jonka kanssa toteutimme diplomityöni puitteissa tehdyn Container App -sovelluksen. Jaakko osallistui diplomityöprojektiini omalla vapaa-ajallaan, koska hän halusi kehittää ammattitaitoaan ja opiskella uusimpia ohjelmointiteknologioita. Jaakolla on pitkä kokemus ohjelmointityöstä ja hänen kanssaan on todella hienoa työskennellä.

Viimeisimpinä mutta ei missään tapauksessa vähäisimpänä haluan kiittää vaimoani ja tytärtäni jotka ovat tukeneet minua ja mahdollistaneet minulle opiskelun vaatimat kotiolot. Kiitokset myös kaikille muille jotka ovat kannustaneet minua opintojeni aikana.

Vaikka minulla oli insinööriopintojen jälkeistä työuraa takana jo noin 17 vuotta ennen diplomi-insinööriopintojen aloittamista niin on hienoa todeta miten hyvällä motivaatiolla pystyin suorittamaan opintoni alusta loppuun. Tästä kokemuksesta jäi hienot muistot ja haluan kannustaa myös kaikkia muita opiskelemaan uuden tutkinnon vaikka työuraa olisikin takana jo hyvän matkaa.

Raumalla, 13.11.2017

Antti Blåfield

SISÄLLYSLUETTELO

1. JOHDANTO

1.1 Euroports Rauma Oy

Diplomityöni toimeksiantaja on Euroports Rauma Oy ja työn tarkoituksena oli selvittää millainen olisi tehokas ja toimiva mobiiliratkaisu merikonttien kunnon ja laadun valvontaan. Merikontit voivat vaurioitua ja likaantua kuljetuksissa ja satamissa ja tarvitsevat toisinaan korjaus- ja pesutoimenpiteitä. Rauman satamassa satamaoperaattori Euroports teettää merikonttien pesu- ja huoltotoimenpiteet alihankintana Arctic Container Oy:llä. Arctic Container Oy on konttialan yritys, joka tarjoaa konttien myynti-, korjaus-, sekä varustelupalveluita. Vuonna 2016 Euroports tarkasti Rauman satamassa noin 50 000 merikonttia. Pesu- ja korjaustoimenpiteitä näihin tarkistettuihin merikontteihin kohdistettiin noin 20 000 kertaa.

Rauman satamaan saapuvissa konttialuksessa voi olla lastina muutamasta sadasta noin 1 500:aan konttia. Kontit saapuvat Rauman satamaan yleensä tyhjinä ja ne tarkistetaan automaattisesti, konttien säilytyssopimukseen perustuen. Silloin, kun kontin korjauksen tai pesun tarve havaitaan niin Euroports tekee siitä arvion kontin omistajalle. Arviossa kerrotaan tarvittavat huoltotarpeet kustannuksineen ja arvio lähetetään omistajalle hyväksyttäväksi. Hyväksynnän saavuttua tieto merkitään Europortsin järjestelmiin ja kontille tehdään tarvittavat toimenpiteet. Keskimäärin luvan saanti kontille kestää noin 4 päivää. Joskus kontti saattaa odottaa lupaa muutaman viikon, mutta joskus lupa tulee jopa samana päivänä kuin arvio on lähetetty. Konteille ei tehdä pesua tai korjausta jos kontin omistaja ei ole sitä hyväksynyt. Suurin osa konttien omistajista on varustamoja ja kontinvuokrausyrityksiä. Konttien liikkeet maailmalla siirtyvät varustamojen tietoon pääsääntöisesti reaaliaikaisesti sanomaliikenteellä.

Rauman satamaan saapuneisiin tyhjiin kontteihin lastataan pääasiassa paperia, sellua ja sahatavaraa. Kontit lähtevät sitten taas lastattuina konttialusten kyydissä pois Raumalta. Metsäteollisuustuotteista, sahatavaran osuus viennistä on noin 25 %, paperin noin 37,5 % ja sellun noin 37,5 %.

Rauman satamassa on meneillään laituri- ja väylälaajennus, joiden valmistuttua satamaan voi tulla suurempia aluksia. Rauman satamassa vierailevien konttialusten koko voi kasvaa niin, että ne voivat kuljettaa kerralla arviolta noin 2 500:aa konttia. Molemmat laajennushankkeet valmistuvat vuoden 2017 loppuun mennessä.

Konttien kunnon ja laadun valvontaan on tarvetta muun muassa konttien vaurioiden korjausten ja pesutoimenpiteiden tarpeen ilmoittamiseen konttien omistajille sekä

mahdollisten korvausvastuiden selvittämiseksi. Konttien pitää olla täysin vesitiiviitä ja puhtaita, jotta kuljetettava lasti säilyy vahingoittumattomana kuljetuksen ajan. Konteista talletettuja tietoja voidaan käyttää myöhemmin myös konttien kunnon ja laadun seurantaan. Tällä hetkellä Europortsilla tai Arctic Containerilla ei ole käytössä konttien valokuvaukseen perustuvaa kunnon- ja laadunvalvontajärjestelmää. Konttien kunnon ja laadun valvontaan ja tietojen tallennukseen tiedostetaan liittyvän erilaisia haasteita. Haasteita ovat muun muassa kohdealueen sääolot ja valaistus. Konttien kunnon ja laadun valvontaan suunniteltavan ratkaisun tulisi myös olla helppokäyttöinen ja toimintavarma vaikeissakin olosuhteissa.

Euroports Rauma Oy on täyden palvelun satamaoperaattori joka toimii Rauman satamassa. Yhtiön toimialaan kuuluvat kaikki satamaoperoinnissa tarvittavat palvelut: lastinkäsittely, varastointi, huolinta, varustamopalvelut, kansainväliset kuljetukset ja tullivarastopalvelut. Alla ilmakuva Rauman satamasta (kuva 1).

Kuva 1. *Ilmakuva Rauman satamasta (Rauman satama)*

1.2 Kontti

Kontit (kuva 2) ovat tavarankuljetussäiliöitä, joilla kuljetetaan tavaroita ympäri maailmaa. Kontteja käytetään maantie-, rautatie-, lento- ja merikuljetuksissa. Kontteja voidaan siirtää eri kuljetusvälineisiin ilman, että sisältöä pitää lastata uudelleen. Konttia käytetään rahtitavaraliikenteen mittayksikkönä ja se vastaa kahdenkymmenen jalan pituista konttia. Kontteja käytetään kuljetustarkoituksen lisäksi myös muunmuassa toimisto- ja työtiloina.

Kuva 2. *Kontti. (Gazouya-japan, Wikimedia Commons, CCo)*

Kontti otettiin käyttöön vuonna 1956 ja Cudahyn (Cudahy, 2006) mukaan konttia pidetään yhtenä tärkeimmistä logistista innovaatioista 1900-luvulla. Vuonna 2009 noin 90 prosenttia merikuljetusten irtotavaralastista kuljetettiin konteissa (Ebeling, 2009). Meriliikenteestä yli 60 prosenttia on konttiliikennettä. Konttialuksia oli vuonna 2016 yli 5 000 kappaletta (Statista, 2017) ja ne pystyvät kuljettamaan kontteja yhteensä 20 miljoonaa kappaletta (Statista, 2017).

Kontit valmistetaan useimmiten teräksestä tai alumiinista. Konttiliikenteessä perusmittayksikkönä on TEU (twenty foot equivalent unit). Yksi TEU tarkoittaa konttia joka on on pituudeltaan 20 jalkaa, leveydeltään 8 jalkaa ja korkeudeltaan 8,5 jalkaa. Kontit on suunniteltu niin, että niitä on helppo pinota ja siirtää sekä ne ovat helposti kiinnitettävissä kuljetusvälineeseen. ISO-kontti on yleisin konttityyppi ja se on kansainvälisen standardisoimisjärjestön (ISO) määrittelemä. ISO-kontti on maailmassa käytetyin konttityyppi, koska se on suurista tuotantomääristä johtuen halpa kuljetusyksikkö. Lentorahdille on käytössä omat konttityypit.

Kontit tekevät vuosittain arviolta 200 miljoonaa matkaa ja kontteja arvioidaan koko maailmassa olevan noin 43 miljoonaa kappaletta (Consultantsea Ltd, 2017). Kiinassa tapahtuu noin 25 prosenttia konttiliikenteestä ja Shanghaissa sijaitsee maailman suurin konttisatama (World Shipping Council, 2017). Konteista myös 95 prosenttia valmistetaan Kiinassa. Suurin konttivarustamo on Maersk Line (Wallenius, 2016). Kuvassa 3 on konttien kuljetukseen käytettävä alus Rauman satamassa.

Kuva 3. *Konttialus konttiterminaalissa Rauman satamassa. (Rauman satama)*

Kontteja kuljetetaan meriliikenteessä konttialuksilla. Konttialuksissa rahti on pakattuna standardikokoisiin kontteihin ja kontit lastataan aluksen ruumaan tai avoimeen lastitilaan. Kontit lastataan laivan rahtitilaan vierekkäin moneen kerrokseen. Konttialuksiin mahtuu paljon lastia ja ovat sen vuoksi parhaimmillaan pitkillä merimatkoilla. Konttialukset liikennöivät sellaisten satamien välillä joissa on konttien käsittelyyn tarvittava laitteisto. Konttialuksissa ei itsessään ole laitteistoa lastin lastaamiseen ja purkamiseen. Maailman suurimmat konttialukset voivat kuljettaa kerralla yli 18 000 kappaletta standardikontteja.

1.3 Tutkimusongelma

Diplomityössä päätutkimuskysymyksen aiheeksi on asetettu suunnitteilla olevan merikonttien kunnon ja laadun valvontaan tarkoitetun tietojärjestelmän arkkitehtuurin syntyminen ketterässä ohjelmistokehityksessä.

Arkkitehtuurin perinteinen tehtävä on ohjelmiston rakenteen ja luonteen selvittämisessä. Arkkitehtuuripohjaisessa ohjelmistokehitysprosessissa arkkitehtuuri määritellään täsmällisesti ohjelmistokehitysprosessin alkuvaiheessa ja sitä noudatetaan tarkasti yksityiskohtaisen suunnittelun ja toteutuksen pohjana. Arkkitehtuurisuunnittelu etenee vaiheittain ja tarkentuu suunnittelun edetessä. Aluksi selvitetään mikä on kohdealue ja mitkä ovat sen vaatimukset. Tämä on niin sanottu asiayhteysvaihe ja tässä selvitetään

mitkä ovat kohdealueen rajat ja myös lähtötiedot koko arkkitehtuurin suunnittelulle. Tässä vaiheessa vastataan myös kysymykseen miksi tehdään.

Käsitteellisessä vaiheessa selvitetään kohdealueen käsitteellinen rakenne ja pyritään vastaamaan siihen mitä kohdealueella tehdään. Tämän jälkeen ratkaistaan loogisella ja välineriippumattomalla tasolla miten toiminnallisuus kohdealueella toteutetaan. Viimeiseksi selvitetään millä teknisillä välineillä loogisesti kuvattu rakenne toteutetaan. Tämä on niin sanottu fyysinen vaihe ja tässä ratkaistaa millä välineillä työ tehdään.

Tässä diplomityössä käytetään tutkimusmenetelmänä toimintatutkimusmenetelmää (Action research). Rapoport (1970) on määritellyt toimintatutkimusmemetelmän seuraavasti: "Toimintatutkimuksella pyritään vaikuttamaan välittömässä ongelmatilanteessa olevien ihmisten käytännön ongelmiin ja yhteiskuntatieteellisiin tavoitteisiin yhteistyössä ja yhteisesti hyväksyttävissä eettisissä puitteissa". Tämä kaksitahoinen näkemys toimintatutkimuksen tavoitteista − asiakkaan ongelman ratkaisemiseksi ja tieteellisen kehityksen edistämiseksi − on kenties toimintatutkimuksen perustavanlaatuisin piirre (Baskerville & Wood-Harper, 1998).

Toimintatutkimus pyrkii myötävaikuttamaan käytännön ongelmiin ja se on siksi joskus sekoitettu sovellettuun tutkimukseen tai konsultointiin (Jönssön, 1991). Toimintatutkimus voi kuitenkin noudattaa tiukkoja ohjeita. Yksi toimintatutkimuksen pääsuuntaviivoista on, että tutkijoiden tulisi tehdä perustelut selkeästi ja organisoida ne siten, että ne ovat testattavissa (Argyris, 1982).

Tämä työ rakentuu seuraavasti. Diplomityön toisessa luvussa esitellään ohjelmistokehitysmenetelmiä, perinteistä vesiputousmallia ja ketteriä menetelmiä. Perinteistä vesiputousmallia ja ketteriä menetelmiä pidetään usein vastakkainasetteluna. Luvusssa 2 perehdytään molempien menetelmien hyviin ja huonoihin puoliin.

Kolmannessa luvussa perehdytään siihen mitä on arkkitehtuuri ohjelmistokehityksessä. Mikä on arkkitehtuurin määritelmä ja tehtävät sekä mitä arkkitehtuuri tarkoittaa ohjelmistokehityksessä. Mitä tarkoittaa arkkitehtuurikuvausten luokittelu ja mitä on arkkitehtuuripainotteinen ohjelmistokehitys. Lisäksi perehdytään arkkitehtuuriin ketterässä ohjelmistokehityksessä ja lopuksi katsotaan miten arkkitehtuureja arvioidaan.

Neljännessä luvussa on dokumentoitu diplomityön puitteissa toteutetua Container App -sovellusta. Sovelluksen toteutuksessa keskityttiin toteuttamaan mobiili- ja web-käyttöliittymää sen verran, että työn toimeksiantajan oli mahdollista arvioida olisiko uusi järjestelmä mahdollista toteuttaa kyseisillä välineillä ja teknologioilla.

2. OHJELMISTOKEHITYSMENETELMÄT

2.1 Vesiputousmalli

Ohjelmiston toteuttaminen on usein laaja ja monimutkainen prosessi. Onnistuneen toteutuksen varmistamiseksi ohjelmistonkehitystyö jaetaan osiin ja seuraavaan osaan voidaan edetä vasta, kun edellinen osa on saatu tarpeeksi valmiiksi. Tällaista ositusta ohjelmistojen suunnittelussa kutsutaan vaihejakomalliksi. Vaihejakomallilla voidaan kuvata myös tuotteen koko elinkaari. Ohjelmiston elinkaari on aika, joka alkaa, kun ohjelmiston kehittäminen aloitetaan ja loppuu, kun ohjelmisto poistetaan käytöstä. Tavallisin vaihejakomalli on niin kutsuttu vesiputousmalli (Haikala & Märijärvi, 2006).

Vesiputousmalli on vaiheellinen ohjelmistotuotantoprosessi, joka sisältää suunnittelu- ja toteutusprosessit. Vesiputousmalli on saanut nimensä siitä, että prosessin vaiheet soljuvat vaihe vaiheelta eteenpäin, kuin vesiputouksessa. Vesiputousmallin ajatuksena korostaa nimenomaan varhaisen suunnittelun merkitystä ohjelmistotuotannossa ja se koostuu lähteestä riippuen 5-7 peräkkäisestä vaiheesta. Vaiheistuksen periaate säilyy kuitenkin muuttumattomana vaiheiden lukumäärästä riippumatta.

Winston W. Royce (1929-1995) kirjoitti 1970-luvulla julkaistun artikkelin (Royce, 1970) josta vesiputousmallin katsotaan saaneen alkunsa. Royce ei kuitenkaan itse käyttänyt tekstissään prosessista termiä "vesiputous" ja vaikka mallista muotoutui ohjelmisto-tuotantomalli, niin itse asiassa hän esitti kyseisen mallin esimerkkinä virheellisestä ja toimimattomasta ohjelmistokehitysmallista. Vesiputousmallissa vaaditaan, että yksi vaihe toteutetaan aina valmiiksi asti ennen, kuin seuraava vaihe voidaan aloittaa. Iteroinnin puuttuminen ja yhden vaiheen valmiiksi tekeminen ennen seuraavan aloittamista eivät olleet Roycen alkuperäisen ajatuksen mukaisia. Itseasiassa Royce piti taaksepäin iterointia tärkeänä.

Roycen (1970) alkuperäisessä vesiputousmallissa oli seuraavat vaiheet:

1. Järjestelmävaatimukset (System Requirements)
2. Ohjelmistovaatimukset (Software Requirements)
3. Analyysi (Analysis)
4. Suunnittelu (Program Design)
5. Ohjelmointi (Coding)
6. Testaus (Testing)
7. Käyttöönotto (Operations)

Vesiputousmallissa vaaditaan, että yksi vaihe toteutetaan aina valmiiksi asti ennen, kuin seuraava vaihe voidaan aloittaa. Esimerkiksi, kun vaatimusten määrittely on saatu

valmiiksi niin siirrytään suunnitteluvaiheeseen, eikä vaatimuksia enää muuteta. Tosin on olemassa päivitettyjä malleja, joissa tätä ohjetta voidaan rikkoa.

Vesiputousmallista on kehitetty vuosien varrella paljon erilaisia variaatioita. Haikalan ja Märijärven kirjassa Ohjelmistotuotanto on esitetty yksi esimerkki vesiputousmalista (kuva 4).

Roycen (1970) mukaan vesiputousmalli on ohjelmistotuotannon peräkkäinen malli, jossa ohjelmiston kehitysvaiheet ovat järjestyksessä ja, kun yksi vaihe valmistuu se dokumentoidaan ja suoritetusta vaiheesta tulee tulevan vaiheen syöte. Vaatimusanalyysi esimerkiksi määrää rajat ja vähimmäisvaatimukset toiminnalliselle määrittelylle. Tällöin määrittelyn lopussa on mahdollista tarkastaa onko ohjelmisto määritelty vaatimus-analyysin mukaisesti. Ohjelmointivaiheessa taas pitäisi olla tekninen määrittely joka kattaa ne tarvittavat tiedot, joiden perusteella ohjelmisto voidaan toteuttaa.

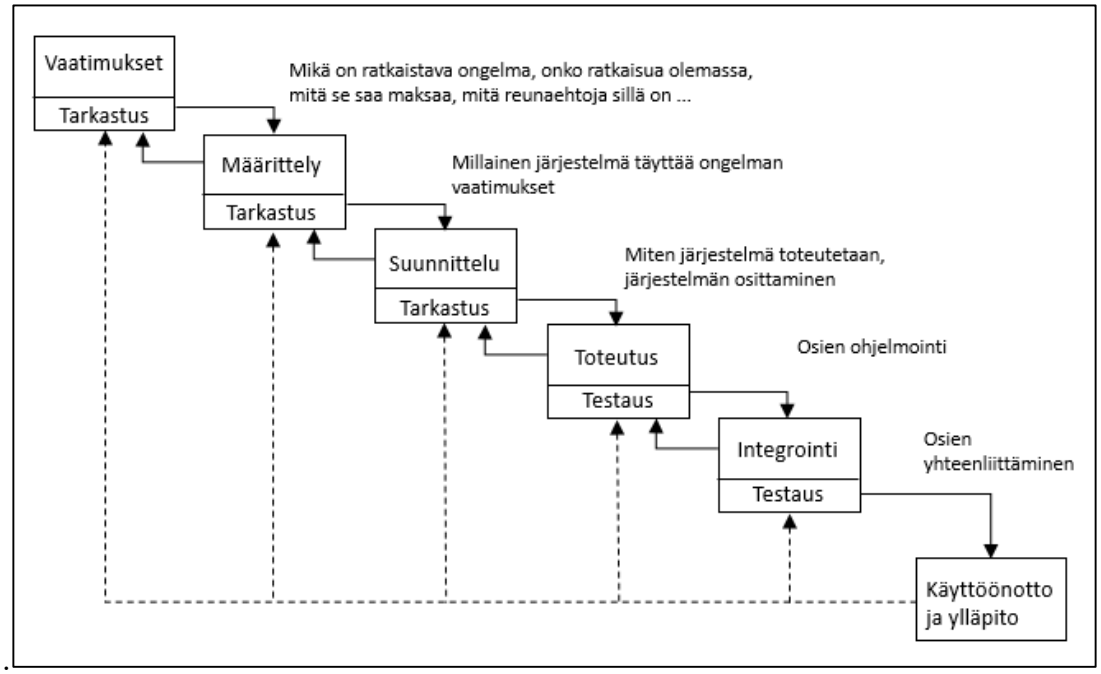
Vesiputousmallissa suurimpana haasteena on suunnitella koko ohjelmistotuote kerralla niin, että se voidaan toteuttaa. Tuotantoprosessi on käytännössä usein iteratiivinen. Iteratiivinen prosessi tarkoittaa sitä, että suunnittelua ja toteutusta tehdään pienimmissä osissa ja prosessia toistetaan. Iteratiivisessa prosessissa ohjelmisto kehittyy inkrementaalisesti eli koko ajan kasvaen kohti lopullista muotoaan. Tämän tarpeen pohjalta ovat syntyneet useat iteratiiviset prosessimallit, kuten esimerkiksi Scrum ja Extreme Programming.

Vesiputousmallin etuna nähdään se, että ohjelmiston huolellinen suunnittelu sen elinkaaren alussa voi johtaa suuriin säästöihin projektin myöhemmissä vaiheissa. On arvioitu, että virhe joka löydetään hyvissä ajoin (esimerkiksi vaatimusten määrittelyssä tai suunnittelussa) on työmäärällisesti, ajallisesti ja taloudellisesti edullisempi korjata, kuin jos sama virhe havaittaisiin myöhemmässä vaiheessa. McConnell (1996) on arvioinut, että jos vaatimusmäärittelyvirhe havaitaan vasta implementaatio- tai ylläpitovaiheessa, niin sen korjaaminen on 50-200 kertaa kalliimpaa, kuin jos virhe olisi huomattu jo vaatimusmäärittelyvaiheessa. Vaatimusmäärittelyvirheen kustannuksista on on myös muita näkemyksiä. Muun muassa Laurent Bossavint (2013) on kirjoittanut yllä mainittujen vaatimusmäärittelyvirheen aiheuttamien kustannusten olevan liioiteltuja.

Kaikkiin vesiputousmallin vaiheisiin liittyy toimenpiteitä laadun varmistamiseen, kuten tarkastuksia, katselmuksia ja testausta. Tarkastuksilla ja testauskella pyritään poistamaan virheet järjestelmästä mahdollisimman varhaisessa vaiheessa. Katselmuksia pidetään yleensä vaiheiden lopuksi. Niissä tilaisuuksissa todetaan projektin tilanne ja se, että kaikki vaiheen tavoitteet on saavutettu ja kaikki sovittu dokumentaatio on tuotettu (Haikala & Märijärvi, 2006).

Vesiputousmallin hyötynä nähdään myös sen keskittyminen kattavaan dokumentointiin. Vesiputousmallin mukaan jokainen vaihe on täydellisesti ja loppuun asti suunniteltu ja

niin esimerkiksi uuden työntekijän on helppo tulla projektiin mukaan missä vaiheessa tahansa, sillä dokumentaatiosta löytyy täydellinen tieto aiemmasta prosessista



Kuva 4. *Esimerkki vesiputousmallista (Haikala & Märijärvi, 2006).*

Vesiputousmalli saa osakseen myös paljon kritiikkiä (Larman, 2004). Vesiputousmallia kritisoidaan yleisesti siksi, että ei uskota olevan mahdollista suunnitella täydellisesti etukäteen mitään suurempaa ohjelmistoa niin, ettei taaksepäin iterointiin olisi tarvetta. Kun asiakas tilaa monimutkaisen ohjelmiston niin se ei pysty määrittelemään omia vaatimuksiaan tarkasti ennen, kuin on päässyt kokeileman prototyyppiä joka toimii jollain tasolla. Asiakas muuttaa myös usein ohjelmiston vaatimuksia kesken projektin, ja vesiputousmallissa tämä tarkoittaa sitä, että iso osa tehdystä työstä, joka on käytetty alussa suunnitteluun, joudutaan tekemään uudelleen. Dokumentoinnin osuus nähdään myös usein liian raskaana toimintona muihin toimintoihin nähden.

Toisaalta suunnittelijat eivät aina osaa ennakoida kaikkia toteutukseen liittyviä ongelmia ennen toteuttamisen aloittamista, ja uuden ratkaisumallin käyttäminen kesken projektin vaatisi uutta suunnittelua. Projektin alussa voidaan esimerkiksi valita toteutukseen teknologioita joilla haluttujen ominaisuuksien toteutus ei onnistu. Lisäksi aiemmissa vaiheissa mahdollisesti tehdyt virheet saattavat muodostua suuriksi ongelmiksi myöhemmissä vaiheissa.

Dean Leffingwellin (2007) kirjassa ”Scalling Software Agility, Best Practices for Large Enterprises”, on luku jossa kerrotaan syitä siihen miksi vesiputousmalli ei toimi. Kirjassa on esitetty neljä keskeistä olettamusta vesiputousmallissa jotka osoittautuvat

virheellisiksi, kun katsotaan asoita siinä valossa mitä olemme ohjelmistokehityksessä oppineet muutaman viime vuosikymmenen aikana.

Vesiputousmallin taustalla olevat olettamukset:

1. On olemassa melko hyvin määritellyt vaatimukset jos vaan käytämme aikaa niiden ymmärtämiseen.

2. Kehittämisprosessin aikana vaatimusten muutokset ovat riittävän pieniä, jotta pystymme hallitsemaan niitä ilman, että suunnitelmaamme uudistetaan tai tarkistetaan.

3. Järjestelmäintegraatio on asianmukainen ja välttämätön prosessi, ja voimme kohtuudella ennustaa, miten se menee arkkitehtuurin ja suunnittelun perusteella.

4. Ohjelmistoinnovaation sekä sellaisen tutkimuksen ja kehittämisen, joka tarvitaan merkittävän uuden ohjelmistosovelluksen luomiseen, voidaan tehdä ennustettavissa olevassa aikataulussa.

Seuraavassa muutamia asioita jokaiselle olettamukselle, jotka osoittavat, että olettamus ei pidä paikkaansa:

Oletus 1: Kuvittelemme ratkaisuja, jotka ajattelemme toimivan; Asiakkaat ovat samaa mieltä siitä, että visio näyttää järkevältä – samalla kun he näkevät samanaikaisesti jotain hieman erilaista. Leffingwell ja Widrig (2003) ovat aikaisemmin kirjoittaneet "Kyllä, Mutta" -oireyhtymästä. Leffingwell kuvaa "Kyllä, Mutta" -oireyhtymää reaktioksi, jonka useimmat meistä kuulevat, kun toimitamme ohjelmistoja asiakkaalle ensimmäistä kertaa: "Kyllä, näen sen nyt, mutta ei, se ei ole juuri sitä mitä tarvitsen". Käytännössä siis määritellyt vaatimukset eivät ole olleet sitä mitä asiakas on oikeasti luullut saavansa.

Oletus 2: Käytännössä ohjelmiston vaatimukset tulevat aina muuttumaan jonkin verran ohjelmiston kehityksen aikana. Jos kehitys oli todella nopeaa ja muutokset olivat todella pieniä, niin tilanne on hallittavissa. Mutta jos kehitys on hidasta ja muutos tapahtuvat nopeasti, niin ollaan vaikeuksissa.

Oletus 3: Tämä olettama on lähtökohtana siitä, että hyvän suunnittelun ja analyysien avulla voimme ennustaa, kuinka monimutkaisen järjestelmän kaikki osat sopivat yhteen ja toimivat keskenään. Väärät olettamukset suunnittelussa johtavat kuitenkin usein suuriin muutoksiin integraatiovaiheessa ja aiheuttavat näin ollen viivastyksiä projektiin.

Oletus 4: Tämä olettama lähtee siitä, että huolellisella suunnittelulla saisimme tarpeeksi tietoa järjestelmästä, sen vaatimuksista ja resursseista, jotta voisimme rakentaa järjestelmän, ja että voisimme luotettavasti ennustaa, milloin aiomme järjestelmän toimittaa. Olettama voi toimia jos kaikki toteutuksessa tehdään aina ensimmäisellä yrityksellä oikein. Toteutuksessa ei kaikki kuitenkaan mene näin ja asioita joudutaan korjaamaan ja tekemään uudelleen. Asioiden uudelleen suunnittelu korjaaminen aiheuttavat sen, että suunniteltu aikataulu ei tule toteutumaan.

Vähän uudemmat projektinhallintamenetelmät pohjautuvat usein myös jollain tasolla vesiputousmalliin. Esimerkiksi ketterä Scrum-sovelluskehitysmenetelmä voidaan nähdä sarjana lyhyitä vesiputousmalleja (Haikala & Mikkonen, 2011).

2.2 Ketterät menetelmät

Takavuosina ajattelutapana on ollut, että varhainen suunnittelu on onnistuneen ohjelmistoprojektin edellytys. Tällainen ohjelmistoarkkitehtuurin täydellinen etukäteensuunnittelu ja ohjelmistoarkkitehtuurin tarkka määrittäminen on saanut monet ohjelmistokehittäjät vastustamaan suunnittelua ja dokumentointia. Ketterät menetelmät (agile software development) syntyivätkin vastareaktiona perinteisille suunnittelu-pohjaisille ja dokumentaatiokeskeisille menetelmille. Vahvasti suunnittelupohjaisissa menetelmissä ohjelmistoarkkitehtuuri voidaan mieltää ohjelmiston perustuslaiksi, jota voidaan muuttaa vain erittäin painavilla perusteilla (Koskimies & Mikkonen, 2005), kun taas ketterät menetelmät ovat luotu vastaamaan nopeasti muuttuviin asiakkaan tarpeisiin ja vaatimuksiin.

Vuonna 2001 17 merkittävää kevyiden ohjelmistokehitysmenetelmien puolestapuhujaa kokoontuivat Utahissa, Yhdysvalloissa, keskustelemaan menetelmiensä yhteisestä perustasta. Heidän tarkoituksenaan oli luoda yhteistä pohjaa ketterille menetelmille ja edistää näin ketterän ajattelun leviämistä. Tuon tapaamisen tuloksena he julkaisivat julistuksen nimeltä Agile Manifesto ("Ketterä manifesti") (Beck, et al., 2001), jota pidetään ketterän kehityksen perusmääritelmänä. Manifestissa on määritelty ketterille menetelmille neljä tyypillistä arvoa ja 12 periaatetta joita menetelmät noudattavat.

Manifestin (Beck, et al., 2001) sisältö on seuraava:

Me etsimme parempia keinoja ohjelmistojen kehittämiseen tekemällä sitä itse ja auttamalla siinä muita. Tässä työssämme olemme päätyneet arvostamaan

Yksilöitä ja vuorovaikutusta enemmän kuin prosesseja ja työkaluja,

Toimivaa sovellusta enemmän kuin kokonaisvaltaista dokumentaatiota,

Asiakasyhteistyötä enemmän kuin sopimusneuvotteluita,

Muutokseen reagoimista enemmän kuin suunnitelman noudattamista.

Vaikka oikeallakin puolella on arvoa, me arvostamme vasemmalla olevia asioita enemmän.

Seuraavaksi katsomme tarkemmin arvojen sisältöä. Ketterien menetelmen arvoissa voidaan nähdä vastakkainasettelua vahvasti suunnittelupohjaisiin menetelmiin.

Yksilöt ja vuorovaikutus

Ketterän manifestin ensimmäinen arvo on "Yksilöt ja vuorovaikutus prosessien ja työkalujen yläpuolella". Ihmisten arvostaminen prosessien tai työkalujen yläpuolelle on helppo ymmärtää, koska ihmiset vastaavat liiketoiminnan tarpeisiin ja ohjaavat kehitysprosessia. Jos prosessit tai työkalut ohjaavat kehitystä niin kehitystiimi ei vastaa muutoksiin eikä näin ollen myöskään asiakkaan tarpeisiin. Tiimin tehtävänä on sopia yhdessä työtehtävistä ja kehittää myös työmenetelmiään jatkuvasti. Vuorovaikutus on esimerkki siitä mikä erottaa yksilöiden ja prosessien arvostuksen. Yksilöiden tapauksessa viestintä on jatkuvaa ja tapahtuu silloin, kun tarve ilmenee. Prosessin tapauksessa viestintä on aikataulutettua ja vaatii erityistä sisältöä.

Toimiva sovellus

Historiallisesti sovelluskehityksessä on käytetty valtava määrä aikaa ohjelmistotuotteen dokumentoimiseen kehitykselle ja asiakkaalle. Tekniset tiedot, tekniset vaatimukset, tekninen esite, käyttöliittymän suunnitteluasiakirjat, testaussuunnitelmat ja dokumentaatiosuunnitelmat. Ketterä kehitys ei sivuta dokumentaatiota kokonaan, mutta se keventää sitä. Dokumentaation pitää olla sellaisessa muodossa, että kehittäjä saa tarpeelliset tiedot työn tekemiseen ilman, että tartutaan pikkuseikkoihin. Ketterä kehitys dokumentoi vaatimukset käyttäjäkertomuksina, jotka riittävät ohjelmistokehittäjille aloittamaan uuden tehtävän rakentamisen. Ketterä manifesti arvostaa dokumentaatiota mutta arvostaa toimivaa sovellusta vielä enemmän.

Asiakasyhteistyö

Esimerkiksi perinteisessä Vesiputous-kehitysmallissa, asiakkaat neuvottelevat tuotteen vaatimuksista usein hyvin yksityiskohtaisesti ennen työn aloittamista. Tämä tarkoittaa, että asiakas osallistui kehitystyöhön ennen kehityksen aloittamista mutta ei itse prosessin aikana. Ketterä kehitys määrittelee asiakkaan roolin niin, että se tekee yhteistyötä kehityksen kanssa koko kehitysprosessin ajan. Tämä helpottaa kehitystä vastaamaan asiakkaan tarpeisiin. Ketterässä kehityksessä asiakas voidaan ottaa esimerkiksi mukaan demoihin säännöllisin väliajoin tai asiakas voi olla jopa loppukäyttäjänä osana tiimiä ja osallistua kaikkiin kokouksiin varmistaen, että tuote vastaa asiakkaan liiketoiminnan tarpeita.

Muutokseen vastaaminen

Perinteisessä ohjelmistokehityksessä kaikki yksityiskohdat suunnitellaan tarkkaan etukäteen ja suunnitelmat ovat myös perusteena ohjelmiston hinnoittelulle. Tämän johdosta perinteisessä ohjelmistokehityksessä muutokset nähdään helposti kustannuksina joita pitää yrittää välttää. Ohjelmistotuotteen vaatimuksiin tulee lähes aina muutoksia kehityksen aikana ja ketterässä kehityksessä ei oletetakaan, että ohjelmisto voitaisiin määritellä tarkkaan etukäteen. Ketterän kehityksen projektin edetessä asiakas ja kehitystiimi saavat paremman käsityksen siitä miten asiakkaan tarpeet saadaan parhaiten täytettyä ja suunnitelmia tehdään kullakin ajan hetkellä vain niin pitkälle kuin on tarpeellista työn etenemisen kannalta.

Haikala ja Mikkonen (2011) ovat suomentaneet ketterien menetelmien periaatteet seuraavasti:

1. Asiakas on pidettävä tyytyväisenä toimittamalla tälle projektin alusta asti tasaisella tahdilla toimivia ohjelmistoversioita.

2. Vaatimusten muuttuminen tulee hyväksyä projektin aikana, myös myöhäisissä kehitysvaiheissa.

3. Toimivien asiakasversioiden julkaisuväli voi vaihdella muutamasta viikosta muutamaan kuukauteen.

4. Liiketoiminta- ja kehitysrooleissa olevien työntekijöiden tulee työskennellä yhdessä päivittäin koko projektin ajan.

5. Projektit tulee rakentaa motivoituneiden yksilöiden ympärille. Heille tulee antaa ympäristö ja tuki, jota he tarvitsevat työn tekemiseksi. Luota siihen, että he saavat työn tehtyä.

6. Kaikkein tehokkain kommunikointikeino sekä kehitystiimin ja ulkomaailman välillä, että itse kehitystiimissä, on keskustelu kasvokkain.

7. Toimiva ohjelmisto on projektin edistymisen tärkein mittari.

8. Ketterät menetelmät edistävät tasaista kehitystahtia. Projektin parissa työskentelevien henkilöiden tulisi kyetä pitämään työtahti ja -kuorma mahdollisimman tasaisena, eikä ylitöitä kuulu tehdä.

9. Huomion jatkuva kiinnittäminen tekniseen erinomaisuuteen ja hyvään suunnitteluun tehostaa ketteryyttä.

10. Asiat pitäisi aina pyrkiä tekemään mahdollisimman yksinkertaisella tavalla: Minimoidaan vähemmän tärkeät tehtävät.

11. Parhaat arkkitehtuurit, vaatimukset ja suunnitelmat syntyvät itseohjautuvasti organisoituneissa tiimeissä.

12. Tiimin tulee selvittää säännöllisin väliajoin, miten se voisi tulla aikaisempaa tehokkaammaksi ja hienosäätää käytettävää työtapaa ja prosessia sen mukaisesti.

Ketterissä menetelmissä ei oleteta, että asiakas pystyy määrittämään oikeat ja täydelliset vaatimukset ohjelmistolle heti projektin alussa. Ketterässä ohjelmistokehityksessä lähdetään myöskin siitä, että ei ole olemassa yhtä oikeaa tapaa päästä haluttuun lopputulokseen. Tämän vuoksi ketterissä menetelmissä ei yleensä ole tarkkaan määritelty miten missäkin tilanteessa tulisi toimia. Arkkitehtuuriset päätökset ohjelmistoprojektin alussa ovat usein sellaisia, joita on kaikista työläin perua tai refaktoroida myöhemmin ja niitä tilanteita ketterä kehitys pyrkii välttämään. Refaktorointi tarkoittaa prosessia, jossa ohjelman lähdekoodia kirjoitetaan uudelleen siten, että sen toiminnallisuus säilyy, mutta sen sisäinen rakenne paranee. Ketterässä ohjelmistokehityksessä on ideana se, että projektin alussa on vaan pieni määrä ohjelmiston vaatimuksia ja niiden pohjalta luodaan osa toimivaa ohjelmaa. Toteutetun ohjelman avulla asiakas pystyy tarkentamaan ja määrittämään lisää vaatimuksia ohjelmistolle. Korkean tason arkkitehtuuri tulisi ketterässä kehityksessäkin olla olemassa alussa, mutta tarkemmat yksityiskohdat suunnitellaan projektin toteutuksen yhteydessä.

Tunnetuimpia ketteriä menetelmiä ovat Extreme Programming (XP) (Beck & Andres, 2004) ja Scrum (Schiel, 2012). Muita ketteriä menetelmiä ovat mm. DSDM, Crystal Methods, Agile modeling, Adaptive software development, Pragmatic Programming, Feature driven development ja Gilb-EVO.

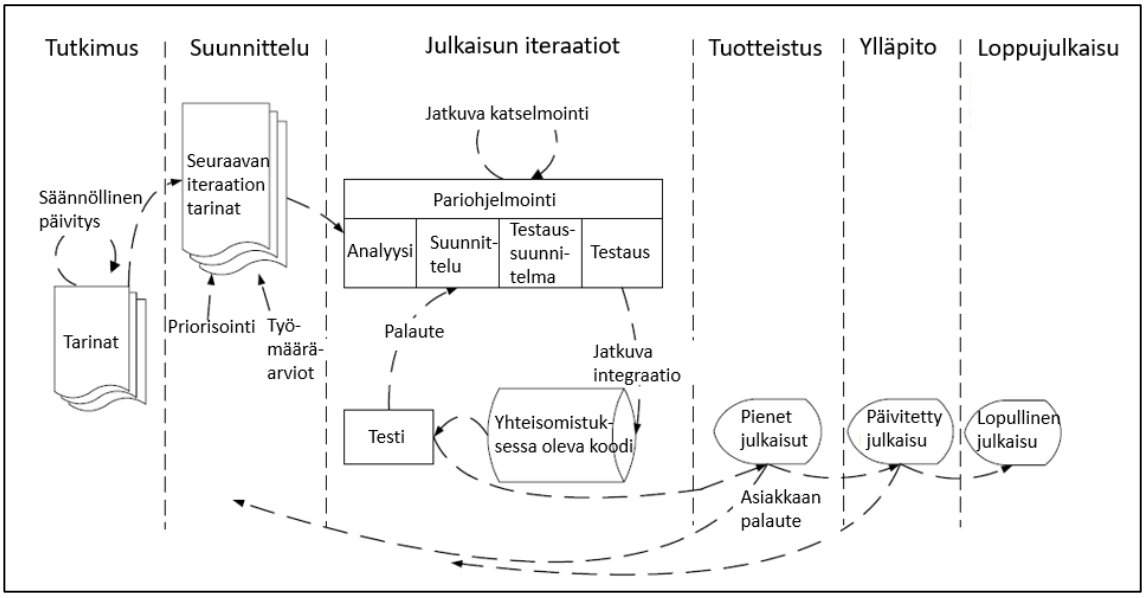
Ketterässä ohjelmistotuotannossa lähtökohtana on jakaa projekti kehitysjaksoihin, joissa keskitytään vain pieneen osaan järjestelmän toiminnallisuutta. Tavoitteena on tuottaa tasaisin väliajoin asiakkaalle ohjelmistoa. Pidemmät projektit jaetaan lyhyisiin tuotantojaksoihin. Scrumissa pyrähdykseksi (sprint) kutsutun jakson suositeltava pituus on 1-4 viikkoa (Schiel, 2012). Extreme Programmingissa jakson pituudeksi suositellaan 1-3 viikkoa (Beck & Andres, 2004).

2.3 Extreme Programming (XP)

Ketterien ohjelmistokehitysmenetelmien alkuna pidetään usein yhdysvaltalaisen Kent Beckin ja hänen kollegoidensa, vuonna 1999, esittelemää Extreme Programming (XP) -menetelmää (Beck, 1999). Kyseessä ei ollut kuitenkaan mikään aivan uusi keksintö vaan Beck kokosi yhteen joukon aikaisempia ideoita ja malleja muodostaakseen niistä oman

teoriansa (Abrahamsson, et al., 2002). Leffingwellin (2007) mukaan XP-menetelmä on todennäköisesti mielenkiintoisin ja ristiriitaisin kaikista ketteristä menetelmistä. Sana "extreme" (äärimmäinen) on menetelmän nimessä Beckin mukaan siksi, että hyväksi havaitut käytännöt on menetelmässä viety äärimmäisyyksiin.

Beckin mukaan ohjelmistokehityksen ongelma on se, että ohjelmistokehitys epäonnistuu ohjelmiston toimittamisessa ja se tarkoittaa sitä, että tällöin ei pystytä myöskään toimittamaan arvoa asiakkaalle. Mahdollisina ongelmina ohjelmistokehityksessä Beck näkee aikataulun venymisen, hankkeen peruuttamisen, ohjelmiston vanhenemisen kehityksen aikana, suuren virheiden määrän ohjelmistossa, asiakkaan liiketoiminnan väärinymmärryksen, asiakkaan liiketoiminnan muutoksen kehityksen aikana, väärien (hienojen) ominaisuuksien toteuttamisen ohjelmistoon ja henkilöstön vaihtuvuuden. Näitä ongelmia XP-menetelmällä halutaan korjata.



Kuva 5. *Extreme Programming -menetelmän eteneminen ja työvaiheet, mukaeltuna (Abrahamsson, et al., 2002).*

XP-menetelmässä toteutus etenee kuuden vaiheen kautta, jotka ovat tutkimus, suunnittelu, julkaisuiteraatiot, tuotteistaminen, ylläpito ja loppujulkaisu. Kuvassa 5 on kuvattu Extreme Programming -menetelmän eteneminen ja työvaiheet.

Extreme Programming (XP) menetelmässä painotetaan muiden ketterien menetelmien tapaan mukautuvuutta enemmän kuin ennustettavuutta. Menetelmän tarkoituksena on parantaa sekä ohjelmiston laatua, että kehitystiimin kykyä vastata asiakkaan vaatimusten muuttumiseen.

XP:n arvoja toteutetaan ja sovelletaan XP:n käytäntöjä noudattamalla. Varsinaisesti XP:n määrittelee kuitenkin XP:n säännöt.

XP-menetelmän arvot:

Kommunikaatio – Tiimin ohjelmistokehityksessä tärkeintä on viestintä. Ongelman ilmetessä kehityksessä, useimmiten joku tietää jo ratkaisun, mutta tieto ei kulje sille, jolla on mahdollisuus tehdä tarvittava muutos. Kysy itseltäsi ongelman ilmetessä, johtuuko ongelma kommunikaation puutteesta. Mitä komunikaatiota tarvitset nyt ongelman ratkaisemiseksi? Minkälaista kommunikaatiota tarvitset jatkossa, että et ajaudu vastaaviin ongelmiin?

Yksinkertaisuus – Yksinkertaisuus on kaikkein intuitiivisin XP:n arvoista. On raskasta työtä tehdä tarpeeksi yksinkertainen järjestelmä joka riittää ratkaisemaan ainoastaan tämän päivän ongelman. Eilinen yksinkertainen ratkaisu voi olla hyvä tänään, tai se voi näyttää liian yksinkertaiselta tai monimutkaiselta.

Palaute – XP:ssa kehitysryhmät pyrkivät tuottamaan niin paljon palautetta, kuin ne pystyvät käsittelemään mahdollisimman nopeasti. Kehitysryhmät yrittävät lyhentää palautesykliä mielummin minuuteiksi ja tunneiksi, kuin viikoiksi ja kuukausiksi. Mitä aikaisemmin tiedät jonkun asian niin sitä nopeammin voit reagoida siihen.

Rohkeus – Rohkeus on tehokas ase pelkoa vastaan. Joskus rohkeus ilmenee ennakoivana toimintana. Jos tiedät, mikä on ongelma, tee sille jotain. Joskus rohkeus ilmenee kärsivällisyytenä. Jos tiedät ongelman olemassaolon, mutta ei tiedä tarkemmin mikä se on, vaatii rohkeutta odottaa todellisen ongelman selkeää ilmaantumista.

Kunnioitus – Edelliset neljä arvoa viittaavat yhteen arvoon, joka sijaitsee muiden neljän arvon pinnan alla ja se arvo on kunnioitus. Jos tiimin jäsenet eivät välitä toisistaan ja siitä mitä toiset tekevät niin XP-menetelmä ei toimi. Jos tiimin jäsenet eivät välitä hankkeesta, mikään ei voi pelastaa sitä. Kaikki ihmiset, jotka ovat tekemisissä ohjelmistokehityksen kanssa, ovat tasavertaisia keskenään. Jotta ohjelmistokehitys parantaisi samanaikaisesti ihmiskuntaa ja tuottavuutta, on jokaisen ihmisen osallistumista tiimiin kunnioitettava.

XP-menetelmän käytännöt on kuvattu Kent Beckin 1999 julkaistussa kirjassa Extreme Programming Explained (Beck, 1999). Yhteenvetona lueteltuihin käytäntöihin Beck kirjoittaa, että yksikään näistä käytännöistä ei toimi yksinään (testausta lukuunottamatta). Käytännöt vaativat toisia käytäntöjä pitämään asiat tasapainossa.

XP-menetelmän käytännöt (Beck, 1999)**:**

- suunnittelupeli (planning game),
- pienet julkaisut (small releases),
- järjestelmän metafora (system metaphor),
- yksinkertainen rakenne ja suunnittelu (simple design),
- testaus (testing),
- uudelleenrakentaminen (refactoring),

- pariohjelmointi (pair programming),
- koodin yhteisomistajuus (collective ownership),
- jatkuva integrointi (continuous integration),
- 40-tuntinen työviikko (40-hourweek),
- paikan päällä oleva asiakas (on-site client) ja
- koodausstandardit (coding standards).

XP-menetelmän säännöt määrittävät varsinaisesti XP:n. Ensimmäisen version XP:n säännöistä julkaisi Don Wells (Wells, 1999) internetsivullaan vuonna 1999. 29 sääntöä on määritelty viidessä eri kategoriassa ja ne ovat iteraation suunnittelu (planning), käytännön hallinnointi (managing), suunnittelu (designing), ohjelmointi (coding) ja Testaus (testing).

XP-menetelmän säännöt (Wells, 1999)**:**

Iteraation suunnittelu (planning):

- käyttäjätarinat on kirjoitettu,
- julkaisun (release) suunnnittelu luo julkaisuaikataulun,
- tee usein pieniä julkaisuja,
- projekti on jaettu iteraatioihin,
- iteraation suunnittelu aloittaa jokaisen iteraation.

Käytännön hallinnointi (managing):

- anna tiimille pysyvä, avoin työtila,
- määritä säilytettävissä oleva työtahti,
- päivä alkaa tapaamisella,
- projektin etenemisnopeutta mitataan,
- henkilöitä kierrätetään projektissa,
- korjataan XP, kun se hajoaa.

Suunnittelu (designing):

- yksinkertaisuus,
- määritellään järjestelmän metafora,
- käytä CRC -kortteja (Class, Responsibilities, and Collaboration),
- luo yksinkertaisia kohdesovelluksia (spike solutions) vähentämään riskejä
- mitään toiminnallisuutta ei lisätä aikaisessa vaiheessa
- refaktoroi aina, kun mahdollista

Ohjelmointi (coding):

- asiakas on aina läsnä,
- koodi kirjoitetaan sovittujen standardien mukaan,
- ohjelmoi yksikkötesti ensin,
- ohjelmakoodi tuotetaan pariohjelmoimalla,
- vain yksi pari kerrallaan integroi koodia,
- integroi usein,
- erillinen tietokone integrointia varten,
- koodi on tiimin yhteisomistuksessa.

Testaus (testing):

- kaikki koodi pitää yksikkötestata,
- kaiken koodin pitää läpäistä kaikki yksikkötestit ennen, kuin versio voidaan julkaistaan,
- kun löydetään virhe, luodaan tarvittavat testit
- hyväksymistestejä ajetaan usein ja tulokset julkaistaan

XP-menetelmästä tekee äärimmäisen se, että se kuvaa innovatiivisia ja kiistanalaisia käytäntöjä ohjelmiston varsinaiseen toteuttamiseen. Itse asiassa myös sanan äärimmäinen, menetelmän nimessä, on tarkoitus herättää niiden huomion, jotka haluavat tutkia tai välttää tämän menetelmän. Beck kirjoittaa, että hänen aikomuksenaan oli tarjota äärimmäisiä ratkaisuja joihinkin pitkäkestoisiin ja läpitunkeviin ohjelmistokehitykseen liittyviin ongelmiin.

3. ARKKITEHTUURI OHJELMISTOKEHITYKSESSÄ

3.1 Arkkitehtuurin määritelmä ja tehtävät

Ohjelmistoarkkitehtuureille on annettu useita erilaisia määritelmiä kirjallisuudessa. IEEE:n (Institute of Electrical and Electronics Engineers, kansainvälinen tekniikan alan järjestö) arkkitehtuurien kuvaamista koskeva standardi määrittelee ohjelmistoarkkitehtuurin järjestelmän perusorganisaatioksi, joka sisältää järjestelmän osat, niiden keskinäiset suhteet ja niiden suhteet ympäristöön sekä periaatteet, jotka ohjaavat järjestelmän suunnittelua ja evoluutiota (IEEE 2000, 2000). Ohjelman tai tietojärjestelmän ohjelmistoarkkitehtuuri on järjestelmän rakenne tai rakenteet, jotka käsittävät ohjelmistoelementtejä, näiden elementtien ulkoisesti näkyviä ominaisuuksia ja niiden välisiä suhteita (Bass, et al., 2003).

Säännöt, joita tietyn arkkitehtuurin mukaan rakennettavassa järjestelmässä on noudatettava, voivat koskea teknologian käyttöä, algoritmien valintaa, tietorakenneratkaisuja, sekä suunnittelu- ja toteutusmalleja. Voidaankin ajatella, että arkkitehtuuri on järjestelmän peruslaki, jota voidaan muuttaa vain erittäin painavilla perusteilla. Arkkitehtuuri pitää dokumentoida selkeästi. Jos arkkitehtuurin dokumentaatiota ei ole olemassa, niin itse arkkitehtuuriakaan ei ole olemassa. (Koskimies & Mikkonen, 2005)

Arkkitehtuuri antaa korkean abstraktiotason näkymän ohjelmistoon, mikä mahdollistaa monimutkaisten järjestelmien tarkastelun ja myös erilaisten sidosryhmien välisen kommunikoinnin ohjelmistosta. Arkkitehtuuri toimii siis yleisenä ohjenuorana ohjelmistokehitykselle ja ohjelmiston ylläpidolle. Arkkitehtuuritason ratkaisut mahdollistavat ohjelmistokehitysprosessissa inkrementaalisen ja rinnakkaisen ohjelmistokehityksen. Inkrementaalisella ohjelmistokehityksellä tarkoitetaan sitä, että ohjelmisto kasvaa koko ajan kohti lopullista muotoaan. Rinnakkainen ohjelmistokehitys taas tarkoittaa sitä, että järjestelmän eri osia toteutetaan saman aikaisesti toisistaan riippumatta. Koskimiehen ja Mikkosen (2005) mukaan arkkitehtuuritasolla tunnistettavat järjestelmän osat ovat mielekkäitä työn jakamisen yksiköitä. Testaamisen kannalta on myös tärkeää tunnistaa järjestelmän osia jotka, voidaan testata erilisinä osina.

Arkkitehtuurin määritteleminen ja dokumentointi mahdollistavat myös järjestelmien arvioinnin kriittisten tekijöiden osalta jo ohjelmistokehityksen varhaisessa vaiheesssa jolloin muutosten tekeminen voi olla vielä yksinkertaisempaa. Pahimmillaan huono arkkitehtuuri voi ilmetä niin, että järjestelmää ei voida toteuttaa suunnitellussa laajuudessa ja aikataulussa. Arkkitehtuuri voi myös tehdä käytettävästä teknologiasta

oletuksia jotka eivät pidä paikkaansa. Mahdollisesti valitulla teknologialla voi olla mahdotonta toteuuttaa esimerkiksi jotain sovelukseen suunniteltuja ominaisuuksia. Huono arkkitehtuuri saattaa aiheuttaa myös sen, että järjestelmä ei pysty suoriutumaan tehtävistään jos käyttäjiä on paljon tai sovelluksen käyttöliittymä voi olla niin hidas, että sitä ei ole mielekästä käyttää.

3.2 Arkkitehtuuri ja ohjelmistokehitys

Ohjelmistotuotannossa arkkitehtuurin tarkoitusta voidaan ajatella monelta kannalta. Arkkitehtuurin perinteinen tehtävä on ohjelmiston rakenteen ja luonteen selvittämisessä. Tämä on erityisen tärkeää silloin, kun ohjelmiston kehityksessä ja ylläpidossa on paljon erilaisia henkilöitä. Arkkitehtuurilla on tässä siis selitävä rooli ja sillä on merkitystä myös sovelluksen ylläpitovaiheessa.

Arkkitehtuurin rooli kasvaa huomattavasti jos se ymmärretään järjestelmän kehitystä ohjaavana artifaktina (Koskimies & Mikkonen, 2005). Tässä käyttötarkoituksessa arkkitehtuuri määritellään täsmällisesti ohjelmistokehitysprosessin alkuvaiheessa ja sitä noudatetaan tarkasti yksityiskohtaisen suunnittelun ja toteutuksen pohjana. Tämän tyyppinen ohjaava arkkitehtuuri johtaa arkkitehtuuripainotteiseen ohjelmisto-kehitysprosessiin.

Arkkitehtuuri voi myös mahdollistaa uuden järjestelmän rakentamisen niin, että samaa arkkitehtuuria käytetään pohjana uudelle ohjelmistolle. Tällainen arkkitehtuurin uudelleenkäyttö mahdollistaa nopeamman ja tehokkaamman tuotekehityksen. Tämän tyyppistä arkkitehtuuria kutsutaankin mahdollistavaksi arkkitehtuuriksi.

Richard F. Schmidt (2013) kuvaa kirjassaan "Software Engineering: Architure-Driven Software Development" ohjelmistoarkkitehtuuria seuraavasti: Ohjelmistotuotteen monimutkaisen luonteen vuoksi on olemassa useita näkökulmia, jotka on ymmärrettävä kuvaamaan ohjelmistotuotteita. Seuraavissa kappaleissa on Schmidtin kolme näkökulmaa ohjelmistotuotteeseen.

Ensimmäisenä asiana arkkitehtuurissa huomioidaan, että ohjelmistotuote on kehitettävä sellaiseksi, että se vastaa asiakkaiden ja sidosryhmien tarpeita ja odotuksia. Tämän vuoksi ohjelmistotuotteen vaatimukset pitää hyväksyttää kaikilla sidosryhmillä ja tallettaa se lähtötasona vaatimuksille. Tämä lähtötaso mahdollistaa myöhemmin muutosehdotusten seurannan lähtötason vaatimuksia vastaan.

Toisena elementtinä ohjelmistoarkkitehtuurissa on toiminnallinen arkkitehtuuri (functional architecture) joka kuvaa ohjelmiston operatiivisen prosessin, toimintojen väliset suhteet, suorituskyvyn ja erikoistekniset ominaisuudet. Ohjelmiston toimintojen väliset suhteet kuvataan siten, että ohjelmisto ikäänkuin palastella helpommin ymmärrettävään muotoon.

Kolmantena elementtinä on ohjelmistotuotteen fyysisen arkkitehtuurin (physical architecture) kuvaaminen rakenteellisesta näkökulmasta. Tämä kuvaa sen miten tuote kootaan ja integroidaan muodostamaan yhden tai useamman ohjelmistokokoonpanon osat. Fyysinen arkkitehtuuri on johdettu toiminnallisesta arkkitehtuurista tavalla, joka sisältää ylhäältä alas käsitteellistämisen ja alhaalta ylöspäin ilmenevän ilmentymisen. Fyysisen arkkitehtuurin perusta on peräisin funktionaalisista yksiköistä, jotka ryhmitellään ja yhdistetään rakenteellisten yksiköiden tunnistamiseksi.

Ilkka Haikala ja Jukka Märijärvi (2006) kirjoittavat kirjassaan Ohjelmistotuotanto seuraavasti. Arkkitehtuurin filosofia antaa kehittäjille mallin siitä miten uusi ominaisuus toteutetaan järjestelmään. Samaten se antaa ylläpitäjälle tietoa siitä minkä tapaista ratkaisua on etsittävä ja mihin muutokset on kohdistettava. Ylipäätään hyvälle arkkitehtuurille on ominaista, että jos jotain asiaa ei tiedä, se on arvattavissa toteutusfilosofian perusteella.

Haikalan ja Märijärven (2006) mukaan onnistunut arkkitehtuuri antaa yleisen helposti kommunikoitavissa olevan mallin, jonka mukaan järjestelmän avainabstraktioita yhdistellään. Onnistunut toteutusfilosofia ja sen yhteydessä käytettävät abstraktiot ovat onnistuneen ohjelmistosuunnittelun avainkysymys, joten suunnittelussa ei kannata edetä ennen, kuin hyvä kokonaisratkaisu on löydetty.

3.3 Arkkitehtuurikuvausten luokittelu ja arkkitehtuuripainotteinen ohjelmistokehitys

Arkkitehtuurikuvaukset voidaan luokitella kolmeen tyypiin niiden ominaisuuksien mukaan (Koskimies & Mikkonen, 2005):

1. käsitteleekö kuvaus rakennetta vai käyttäytymistä;
2. käsitteleekö kuvaus järjestelmän staattisia vai dynaamisia piirteitä; ja
3. määritteleekö kuvaus tietyn asian kokonaisuudessaan vai antaako se vaan esimerkin siitä.

Näiden perusteella voidaan muodostaa kuusi eri kuvaustyyppiä: rakennekuvaus, käyttäytymiskuvaus, staattinen kuvaus, dynaaminen kuvaus, esimerkkikuvaus ja määrittelykuvaus.

Koskimiehen ja Mikkosen (2005) mukaan arkkitehtuuripainotteisessa ohjelmistokehityksessä arkkitehtuurin suunnittelu ja arviointi ovat keskeisinä vaiheina ennen, kuin yksityiskohtainen suunnittelu ja toteutus aloitetaan. Arkkitehtuuriin vaikuttavat keskeisimmät toiminnalliset ja laadulliset vaatimukset. Yleensä arkkitehtuurisuunnittelu etenee niin, että ensin arkkitehtuurista tehdään versio toiminnallisten vaatimusten pohjalta ja sitä arvoidaan laadullisia ominaisuuksia vasten. Tarvittaessa arkkitehtuuria

muutetaan niin, että laadulliset vaatimukset täyttyvät. Sitten, kun kaikki laadulliset vaatimukset on kunnossa niin järjestelmän perusarkkitehtuuri on valmis.

3.4 Arkkitehtuuri ketterässä ohjelmistokehityksessä

Ketterää ohjelmistokehitystä ja arkkitehtuurilähtöistä ohjelmistotuotantoa pidetään usein vastakkainasetteluna. Ketterä ohjelmistokehitys hylkää perinteisen arkkitehtuurisuunnittelun ja järjestelmän rakentamisen lähes kokonaan, sillä tiukka suunnitelmallisuus ei sovi ketteriin arvoihin.

Ketterässä ohjelmistotuotannossa ei tehdä raskasta suunnitteluprosessia aluksi, vaan järjestelmän rakenne muodostuu asiakkaan ja kehitysryhmän valitsemien toteutettavien ominaisuuksien mukaan (Leffingwell, 2007). Ohjelmiston arkkitehtuuri muodostuu inkrementaalisesti kehitysjaksosta toiseen. Ohjelmiston arkkitehtuurin inkrementaalinen muodostuminen tarkoitaa sitä, että arkkitehtuuri kasvaa koko ajan kohti lopullista muotoaan.

Ketterässä kehityksessä pääpaino on siis iteratiivisessa ongelmanratkaisussa. Arkkitehtuuriin liittyviä kysymyksiä pyritään ratkaisemaan sitä mukaa, kun niitä ilmenee projektin edetessä. Vastuuta jaetaan koko tiimille, jolloin jokainen tiimin jäsen pääsee optimitapauksessa vaikuttamaan arkkitehtuuriin ja ymmärtää myös miksi tietyt ratkaisut on tehty. Ketterien sovelluskehitysmenetelmien käyttöönotto on lisääntynyt erittäin paljon kaikenlaisilla yrityksillä viime vuosina (VersionOne, 2017). Tämän johdosta on lisääntynyt myös skeptismi sellaisten ketterien sovelluskehitysmenetelmien luotettavuudesta ja tehokkuudesta, jotka eivät kiinnitä riittävästi huomiota ohjelmistoarkkitehtuurin tärkeisiin periaatteisiin (Babar, et al., 2014).

Suurin osa ketterien ohjelmistokehitysmenetelmien kuvauksista kiinnittää hyvin vähän huomiota yleiseen arkkitehtisuunnitteluun, kuten arkkitehtoniseen analyysiin, arkkitehtoniseen synteesiin, arkkitehtoniseen arviointiin ja näihin toimintoihin liittyviin artefaktityyppeihin (Hofmeister, et al., 2007). Suurin osa ketteristä ohjelmistokehitysmenetelmistä pyrkii olettamaan, että arkkitehtoninen suunnittelu on korkeatasoista suunnittelua ilman selkeitä jäsentelyvoimia, kuten laatuominaisuuksia. Thapparambilin (Thapparambil, 2005) mukaan refaktorointi on ensisijainen menetelmä kehittää arkkitehtuuria ketterässä ohjelmistokehityksessä.

Esimerkiksi Scrum -menetelmässä (Schiel, 2012) arkkitehtuuria ei huomioida juuri ollenkaan mutta Extreme programming (XP) -menetelmän kehittäjä Beck (1999) kirjoittaa: "Arkkitehtuuri on yhtä tärkeä XP-projektissa, kuin missä tahansa muussa ohjelmistoprojektissa." Vaikka XP-menetelmässä yksikään käytännöistä ei nimenomaisesti osoita ohjelmistoarkkitehtuurin roolia niin tämä ei kuitenkaan tarkoita, etteikö XP-menetelmä tai -tiimi ymmärtäisi arkkitehtuurin roolia sovelluskehityksessä.

Beck selittää miten arkkitehtuuri syntyy (Beck, 1999):

Valitse ensimmäiseen iteraatioon joukko yksinkertaisia perustarinoita joiden odotat pakottavan luomaan koko arkkitehtuurin. Sitten kavenna horisonttia ja toteuta tarinat mahdollisimman yksinkertaisella tavalla. Ja tämä harjoituksen lopuksi sinulla on arkkitehtuuri.

Nämä kommentit tarjoavat hyvin lisätietoa XP-näkökulmasta. Jos järjestelmä on sen verran yksinkertainen, niin että pienet tarinat ja iteraatio tai kaksi voivat muodostaa kohtuullisen arkkitehtuurin perustan, tämä lähestymistapa voi olla erittäin tehokas ja arkkitehtuuri voi ilmetä melko hyvin tässä mallissa. Ja koska XP-menetelmää on suositeltu ja sovellettu ensisijaisesti pieniin tiimeihin, sen näkemys tiimin koosta ja arkkitehtuuristrategiasta on johdonmukainen (Leffingwell, 2007).

XP-kirjan (Beck & Andres, 2004) toisen painoksen ensisijainen inkrementaalinen suunnittelukäytäntö väittää, että arkkitehtuuri voi ilmetä päivittäisessä suunnittelussa. Uuden suunnittelun ansiosta arkkitehtuuri tukeutuu potentiaalisesti huonojen arkkitehtonisten ratkaisujen etsimiseen toteutetusta koodista ja paremman arkkitehtuurin luomisesta tarpeen mukaan refaktoroimalla. Tämän lähestymistavan mukaan arkkitehtuuri syntyy koodista pikemminkin, kuin minkäänlaisena etukäisuunnitelmana.

Ohjelmistoarkkitehtuuri voidaan eriyttää ketterässä kehityksessä myös omaksi prosessikseen (Leffingwell, 2007). Tässä toimintatavassa erilliset arkkitehdit suunnittelevat ja mahdollisesti myös toteuttavat arkkitehtuurin. Arkkitehtiryhmällä on oma aikataulu ja he noudattavat omaa prosessimalliaan. Kehitysryhmä määrittelee tarkastuspisteet joihin arkkitehtiryhmä sitoutuu ja tarkistuspisteissä tietty osa järjestelmän rakenteesta täytyy olla valmis kehitysryhmää varten. Arkkitehtiryhmä keskittyy ohjelmiston myöhempien kehitysjaksojen vaatimuksiin ja ryhmän vastuulla on suunnitella järjestelmän rakenne niin, että se tukee seuraavia lisättäviä toimintoja. Arkkitehtuuri eriytettynä prosessina, ketterässä kehityksessä, varaa ohjelmistoarkkitehtuurille huomattavasti ennemmän aikaa, kuin silloin jos arkkitehtuuria ei ole eriytetty.

3.5 Arkkitehtuurien arviointi

Aiemmin esitettiin, että arkkitehtuuri suunnitellaan laadullisten vaatimusten ehdoilla ja näin ollen arkkitehtuuri ratkaisee miten hyvin ohjelmisto täyttää laadulliset vaatimukset. Arkkitehtuurien arvioinnissa päähuomio on siis laadullisten vaatimusten huomioinnissa, ei toiminnallisten vaatimusten huomioinnissa. On tärkeää, että arkkitehtuuri sisältää kaikki olennaiset laatuun vaikuttavat ratkaisut, jotta ohjelmiston laadulliset ominaisuudet saadaan arvioitua kattavasti. Tätä voidaan pitää arkkitehtuurin täydellisyyden kriteerinä: jos jotakin laadullista ominaisuutta ei kyetä arkkitehtuurin peruusteella arvioimaan, arkkitehtuuri on tältä osin puutteellinen (Koskimies & Mikkonen, 2005).

Koskimiehen ja Mikkosen (2005) mukaan yleisiä laatuvaatimuksia ohjelmistoille ovat esimerkiksi:

- **Suorituskyky**; järjestelmän kuluttamat resurssit tietyn data-, tapahtuma- tai käyttäjämäärän käsittelemiseen;
- **Luotettavuus**; järjestelmän kyky pysyä toimintakelpoisena;
- **Saatavuus**; järjestelmän pystyssäoloajan suhteellinen osuus;
- **Turvallisuus**; järjestelmän kyky torjua oikeudettomat käyttäjät aiheuttamatta haittaa laillisille käyttäjille;
- **Muunneltavuus**; muutoksien tekemisen helppous;
- **Siirrettävyys**; kuinka hyvin järjestelmä tukee siirtoaan eri resurssiympäristöihin; ja
- **Joustavuus**; kuinka hyvin järjestelmässä on otettu huomioon tiettyjen vaatimusten vaihtelu.

Kuten aiemmin on todettu, ohjelmistojen arkkitehtuuri on tärkeä ja tämän vuoksi on suositeltavaa arvioida sitä säännöllisesti jo ohjelmistosuunnittelun alkuvaiheessa. Suunnitteluvaiheen arkkitehtonisen muutoksen kustannukset ovat vähäpätöiset verrattuna arkkitehtonisen muutoksen kustannuksiin järjestelmässä, joka on jo toteutusvaiheessa (Jansen & Bosch, 2005). Tämän vuoksi kustannuksia voidaan vähentää arvioimalla ohjelmistokehitysarkkitehtuuria ennen sen toteutusta tunnistamalla riskejä ja ongelmia hyvissä ajoin.

Arkkitehtuurin arvioinnissa arvioidaan muunmuassa komponenttien ja alijärjestelmien suhteita ja niiden ominaisuuksia. Arkkitehtuurin arvioinnissa tutkitaan myös pystyykö järjestelmä täyttämään sille asetetut vaatimukset, myös tulevaisuudessa. Tulevaisuuden vaatimuksiin kuuluvat laajennettavuus, muunneltavuus ja skaalautuvuus ilman, että esimerkiksi suorituskyky tai muistinkulutus kärsivät liikaa.

Arkkitehtuuria voidaan arvioida erilaisilla arviointimenetelmillä ja niillä saadaan yleensä vastauksia seuraaviin kysymyksiin:

- Sopiiko suunniteltu arkkitehtuuri järjestelmälle?
- Mikä vaihtoehtoisista arkkitehtuureista soveltuu parhaiten järjestelmälle ja miksi?
- Miten hyvä tulee olemaan järjestelmän jokin tietty laadullinen ominaisuus?

Arkkitehtuurin arviointimenetelmä voi olla esimerkiksi skenaariopohjainen. Skenaariopohjaisessa arvioinnissa esitetään konkreettisia esimerkkitilanteita, joissa laatuominaisuudet tulevat esiin.

ATAM (Architecture Tradeoff Analysis Method) on hyvin tunnettu skenaariopohjainen arviointimenetelmä. ATAM -menetelmää on käytetty yli vuosikymmenen ajan arvioimaan ohjelmistokehitysarkkitehtuuria aloilla, jotka ulottuvat autoteollisuudesta

rahoitukseen ja puolustukseen. ATAM on suunniteltu niin, että arvioitsijoiden ei tarvitse tuntea arkkitehtuuria tai liiketoiminnan tavoitteita, järjestelmää ei tarvitse vielä rakentaa ja sidosryhmiä voi olla paljon. (Clements, et al., 2012)

Kirjassa ”Economics-Driven Software Architecture” (Mistrik, et al., 2014) painotetaan yrityksen sovellusarkkitehtien ja johdon välistä viestintää arkkitehtuurin taloudellisen suunnittelun onnistumiseksi. Sovellusarkkitehdit tekevät arkkitehtonisia suunnittelu-päätöksiä säännöllisesti, mutta eivät yleensä pysty arvioimaan näiden päätösten taloudellisia vaikutuksia. Johto on sitä vastoin usein kiinnostunut tuotetason päätöksistä, kuten ominaisuuksista ja laadusta, mutta ei teknisissä yksityiskohdissa siitä, miten nämä päätökset tehdään. Nämä erilaiset intressit johtavat epäjohdonmukaisuuksiin siitä, miten johtajat arvostavat arvoa ja miten arkkitehdit voivat ottaa käyttöön tai hylätä nämä arvoehdotukset suunnittelupäätöksensä kautta. Viestinnän puute sovellusarkkitehtien ja johdon välillä voi johtaa huonoihin päätöksiin.

On selvää, että kaikkien hankkeen sidosryhmien etujen mukaista on tehdä tietoon perustuvia ja teknisesti toteuttamiskelpoisia arvopohjaisia suunnittelupäätöksiä. Mistrik, et al. (2014) mukaan arkkitehdit tarvitsevat käytännöllisiä, validoituja työkaluja ja tekniikoita taloustietoihin perustuvien periaatteiden soveltamiseksi ohjelmistoarkkitehtuuriin. He tarvitsevat näitä välineitä ja tekniikoita tekemään parempia päätöksiä ja perustelemaan paremmin kyseiset päätökset sidosryhmilleen.

Ketterät kehittämismenetelmät, kuten Scrum (Schiel, 2012), eivät perusta ohjelmiston arkkitehtuurin suunnittelusta. Agile-manifesti (Beck, et al., 2001) määrittää, että parhaita arkkitehtuureja syntyy tiimeistä. Esimerkiksi Scrum-kehittämismenetelmää käyttävät kehittäjät tuntuvat ajattelevan, että Scrumia käytettäessä ei ole tarvetta etukäteen tehtävään ohjelmiston arkkitehtuurin määrittelyyn tai arkkitehtuurin arviointiin missään kohtaa (Mistrik, et al., 2014). Näin ei kuitenkaan ole. Jos arkkitehtuurin arvioinnista on löydettävissä asiakkaalle arvoa niin arviointi olisi tehtävä. Tämän erilaisen ajatusmaailman ovat arkkitehtuurimaailman ja ketterän maailman välillä on tunnistaneet monet kirjoittajat. Se, että monet suosittut arkkitehtuurien arviointimenetelmät kestävät useita päiviä, kun niitä toteutetaan täysimittaisesti, vahvistaa tätä ongelmaa. Leffingwellillä (2007) on joitakin pyrkimyksiä löytää parhaita käytäntöjä arkkitehtuurityön ja ketterän suunnittelun yhdistämiseen. Hän kuvaa muun muassa arkkitehtonisen kiitoradan (architectural runway) syntymistä. Nämä käytännöt eivät kuitenkaan esitä ratkaisuja arkkitehtuurin arviointiin.

4. CONTAINER APP

4.1 Idea sovelluksen toteuttamisesta

Ehdotus toteuttaa mobiilisovellus merikonttien kunnon ja laadun valvontaan tuli Euroports Rauma Oy:llä. Esittelin Europortsilla työskentelevälle tuttavalleni mobiiliohjelmoinnin kurssilla tekemääni harjoitustyötä, jossa Android -mobiililaitteella otettu kuva ja kuvanottopaikan koordinaatit talletettiin tietokantaan. Tietokantaan talletettuja tietoja pystyi myös selailemaan mobiililaitteella. Tämän esittelyn pohjalta syntyi ajatus siitä, että saman tyyppisesti toteutettua sovellusta voisi käyttää myös muunmuassa merikonttien kunnon ja laadun valvontaan.

Ajatus uuden sovelluksen toteutuksesta syntyi myös siitä, että nykyinen järjestelmä, jota käytetään konttien tarkistukseen ja vaurioiden dokumentoimiseen, on jokseenkin vanhentunut. Nykyisessä järjestelmässä käytettävien käsipäätteiden korvaaminen pienemmillä ja edullisemmilla mobiililaitteilla, esimerkiksi puhelimilla, olisi myös positiivinen asia. Mobiili-käyttöliittymän lisäksi uuteen sovellukseen pitäisi toteuttaa myös web-käyttöliittymä jolla konteista talletettuna tietoja voi tarkastella tietokoneella.

Kokonaan uuden sovelluksen suunnittelu ja toteuttaminen merikonttien kunnon ja laadun valvontaan ja seurantaan olisi laaja työ ja sitä olisi mahdotonta toteuttaa diplomityön puitteissa. Sovelluksen toteutuksessa keskitytäänkin nyt toteuttamaan mobiili- ja web-käyttöliittymää sen verran, että työn toimeksiantajan on mahdollista arvioida olisiko uusi järjestelmä mahdollista toteuttaa kyseisillä välineillä ja teknologioilla. Yleisesti ottaen mobiililaitteella pitäisi kuitenkin saada tehtyä konttien tarkistustoimenpiteet alusta loppuun ilman, että kontin tarkastajan tarvitsee käyttää web-käyttöliittymää.

Toisaalta sovelluksen toteutuksen yhteydessä voidaaan vastata myös tämän työn tutkimustavoitteeseen. Sovelluksen arvionnissa yksi keskeinen asia on se millainen mobiililaitteen käytettävyys on kenttäolosuhteissa. Konttikentällä sääolosuhteet voivat olla hyvinkin haastavat ja sen vuoksi mobiilisovelluksen käyttöliittymän pitää olla visuaalisesti selkeä ja mahdollisimman yksinkertainen käyttää. Kontin sisällä on joko hämärää tai pimeää, ulko-olosuhteiden valoisuudesta riippuen, ja konttien tarkastajilla on lisävalonlähteenä tasku- tai otsalamput.

4.2 Toteutus

Päätimme lähteä toteuttamaan mobiilisovellusta merikonttien kunnon ja laadun valvontaan Extreme Programming (XP) -menetelmää käyttäen (Beck, 1999). Ketterissä menetelmissä ei oleteta, että asiakas pystyy määrittämään oikeat ja täydelliset vaatimukset ohjelmistolle heti projektin alussa. Ketterässä ohjelmistokehityksessä on ideana se, että projektin alussa on vaan pieni määrä ohjelmiston vaatimuksia ja niiden pohjalta luodaan osa toimivaa ohjelmaa. Toteutetun ohjelman avulla asiakas pystyy tarkentamaan ja määrittämään lisää vaatimuksia ohjelmistolle. Ketterässä ohjelmistotuotannossa ei tehdä raskasta suunnitteluprosessia aluksi, vaan järjestelmän rakenne muodostuu asiakkaan ja kehitysryhmän valitsemien toteutettavien ominaisuuksien mukaan (Leffingwell, 2007).

Jo varhaisessa vaiheessa sovellusta hahmoteltaessa selvisi, että sovellukseen pitäisi saada mobiilikäyttöliittymän lisäksi myös taulutietokoneella tai tavallisella tietokoneella käytettävä web-käyttöliittymä. Web-käyttöliittymän toteuttaminen itse olisi kasvattanut diplomityöni aiheen liian laajaksi ja pyysin kollegani mukaan projektiin toteuttamaan web-käyttöliittymää. Kollegani on toiminut työssään pitkään ohjelmointitehtävissä ja hänellä on laaja kokemus eri ohjelmointikielistä ja teknologioista. Hän lähti mukaan projektiin, koska hän halusi tutustua tämän hetken uusimpiin ohjelmointialustoihin joita hän ei normaalisti työssään käytä.

Extreme Programming (XP) -menetelmä valikoitui sovelluksen toteutuksen menetelmäksi vaikka puhdasta XP-menetelmää ei kovin paljon nykyään enää ohjelmistokehityksessä käytetä. XP:ssä huomioidaan kuitenkin ohjelmiston arkkitehtuuria ennemmän (Leffingwell, 2007), kuin monessa muussa ketterän ohjelmistokehityksen menetelmässä ja sillä perusteella tein valinnan. Extreme programming (XP) -menetelmän kehittäjän Beckin (Beck, 1999) mukaan arkkitehtuuri on yhtä tärkeä XP-projektissa, kuin missä tahansa muussa ohjelmistoprojektissa. Lisäksi XP-menetelmä sopii hyvin pienessä ryhmässä sovelluksen toteutukseen. Tiimissä oli kaksi ohjelmoijaa, minä ja kollegani, sekä kaksi työn toimeksiantajan edustajaa Europortsilta.

Selvästi suosituin ketterän kehityksen menetelmä on State of Agile Report -julkaisun (VersionOne, 2017) mukaan Scrum. Raportin mukaan toiseksi suosituin ketterän kehityksen menetelmä on Scrum/XP hybrid. Hyviksi todettuja XP-menetelmän osia käytetään siis myös muissa ketterissä menetelmisssä.

4.3 Järjestelmän vaatimukset

Kävin tutustumassa kohdealueeseen ensimmäisen kerran 31.5.2017 (kuva 6). Ohjattu tutustumiskäynti kesti noin kaksi tuntia ja pääsin tutustumaan satama-alueella konttien tarkastukseen, korjaukseen ja lastaukseen. Seuraavissa alikappaleissa käydään läpi tarkemmin konttien tarkastusta, vaurioiden estimointia sekä korjausta ja pesua.

Kuva 6. *Konttikurottajat siirtävät kontteja satamassa. (Antti Blåfield)*

4.3.1 Konttien tarkastus kentällä

Konttien tarkastajat tarkistavat kontit visuaalisesti sisältä ja ulkoa. Kontin sisällä on joko hämärää tai pimeää, ulko-olosuhteiden valoisuudesta riippuen, ja konttien tarkastajilla on tasku- tai otsalamput käytössään.

Konteista tarkistetaan oviensaranat, ovet, seinät, lattia, katto ja pohja- sekä kulmarakenteet. Konteista etsitään vaurioita sekä tarkistetaan siisteys ja puhtaus. Vauriot voivat olla esimerkiksi painaumia tai reikiä seinissä ja lattioissa. Vauriot ovat voineet tulla esimerkiksi mekaanisesti trukin piikistä tai ruoste on voinut tehdä reiän. Kuvassa 7 näkyy kontin seinässä olevia jälkiä, jotka ovat syntyneet trukin piikistä. Kontin maalipinnan rikkouduttua metalli alkaa ruostua nopeasti meriolosuhteissa.

Kuvan 7 vaurioihin ei todennäkoisesti kohdisteta vielä korjaustoimen-piteitä mutta myöhemmin seinän vauriot joudutaan mahdollisesti oikaisemaan ja maalaamaan.

Kuva 7. *Trukin piikin tekemät vairiot kontin seinässä. (Antti Blåfield)*

Konttien lattia on tehty kovapuuvanerilevyistä jotka joutuvat kovalle koetukselle, kun esimerkiksi paperirullia lastataan kontin sisään trukeilla. Kuvassa 8 näkyy vaurio kontin lattiassa. Vaurio on syntynyt, kun trukki on ajanut kontin sisälle. Vaurio pitää korjata vaihtamalla tarpeellinen määrä lattialevyjä.

Kuva 8. *Kontin lattia on vaurioitunut trukin painosta. (Antti Blåfield)*

Kontit voivat myös likaantua sisältä. Kuvassa 9 kontin lattialla on luultavasti öljyä. Öljy voidaan poistaa lattiasta polttamalla. Polttamisen jälkeen lattia vielä hiotaan.

Kuva 9. *Likaantunut kontin lattia. (Antti Blåfield)*

Jos tarkastuksessa havaintona tai epäilyksenä on, että kontin lattia- tai pohjarakenne on vaurioitunut niin kontti nostetaan pukkien päälle lattian tai pohjan tarkastusta varten. Kuvassa 10 kontti on pukkien päällä tarkastettavana.

Kuva 10. *Kontti pukkien päällä tarkastettavana. (Antti Blåfield)*

Kokemuksen mukaan konteissa esiintyviä hajuja olisi hyvä saada myös dokumentoitua mutta siihen ei ole muuta tapaa, kuin kontin tarkastajan subjektiivinen kuvaus hajusta.

Tyhjien ja käyttökunnossa olevien konttien tarve on saatavilla Europortsin järjestelmistä ja näin ollen konttien tarkistus ja korjaustyöt voidaan rytmittää niin, että käyttökunnossa olevia kontteja on koko ajan tarpeeksi saatavilla. Konttien tarkastuksia tehdään yleensä kahdessa työvuorossa, mutta jos on kiirettä niin tarkastuksia voidaan tehdä kolmessa työvuorossa.

4.3.2 Konttien vaurioiden estimointi

Kontin tarkastaja syöttää vaurioituneesta tai likaantuneesta kontista tiedot käsipäätteen avulla John Evans -järjestelmään. Tietojen syöttämisen jälkeen järjestelmä laskee hinta-arvion (estimaatin) kontin korjaus- ja/tai pesukustannuksista. Hinta-arvio on myös tarvittaessa nähtävissä käsipäätteestä välittömästi tietojen syöttämisen jälkeen. Hinta-arvio lasketaan järjestelmään ennalta syötetyistä tariffeista. Tariffit sisältävät erilaisten vaurioiden korjausten työtuntien tarpeen sekä materiaalien ja työtuntien hinnat.

Käsipäätteeseen syötetään kontin ID-tunniste ja tiedot vauriosta tai likaisuudesta. Tietojen syöttö etenee vaiheittain ja yhdelle kontille voidaan syöttää kerralla useampi vaurio tai likaisuus. Käsipäätteessä on myös kamera jolla vauriokohta tai likaisuuus kuvataan. Sen jälkeen, kun tiedot on syötetty käsipäätteelle niin kontin tarkastaja syötää tiedot myös satamalogistiikan tietojärjestelmä PortOperaan. Tämän jälkeen tiedot menevät RTK:lle, joka laittaa korjaustarjousviestin kontin omistajalle. Korjaustarjousviestin lähetys tapahtuu joko John Evans -järjestelmän automaattisesti luomalla sähköpostilla asiakaskohtaisesti määriteltyihin osoitteisiin, WESTIM (EDI) sanomalla tai manuaalisesti syöttämällä asiakkaan omaan järjestelmään.

PortOperaan syötetään kontin laatuvika (rikki, pestävä jne.) ja luokitus (paperi, sellu, sahatavara jne.) sekä tarvittavat toimenpiteet (korjaus-/pesutapa). Tämän perusteella kontti osataan siirtää oikeaan paikkaan odottamaan toimenpiteitä.

Kontin omistajan hyväksyttyä korjaustarjouksen RTK kirjaa siitä tiedon PortOperaan ja kontti voidaan siirtää Arctic Containerin alueelle pesu- tai korjaustoimenpiteitä varten.

4.3.3 Konttien korjaus ja pesu

Arctic Containerin toimipiste sijaitsee Rauman satamassa konttikentän välittömässä läheisyydessä ja korjaus- ja/tai pesutoimenpiteitä vaativat kontit siirretään Arctic Containerin alueelle konttikurottajilla.

Arctic Container tekee muunmuassa seuraavia korjaustoimenpiteitä konteille:

- Seinän tai katon vaurioituneen kohdan korjaamiseksi päälle hitsataan levy (patch),
- Seinästä tai katosta voidaan myös ensin leikata pala pois ja sen jälkeen hitsataan samankokoinen pala uutta levyä tilalle (insert),
- Jonkin osan vaihto uuteen (replace),
- Jonkin osan uudelleen kiinnitys (resecure),
- Osien suoristus (straighten),
- Hitsaus (welding),
- Maalaus (painting) ja
- Jonkin osan osittainen korjaus (section).

Kontteihin kohdistuvia pesutoimenpiteitä ovat muunmuassa:

- Vesipesu,
- Liuotinpesu ja
- Spottipesu (öljyn polttaminen pois pinnoilta ja poltetut alueen hionta).

Korjaus- ja/tai pesutoimenpiteiden jälkeen kontit siirretään takaisin Europortsin konttikentälle, konttien varastointialueelle, ja odottamaan seuraavaa lastausta.

4.4 Ensimmäinen toteutusjakso

4.4.1 Ensimmäinen suunnittelupeli

Pidimme ensimmäisen suunnittelupelin 1.8.2017 jossa määrittelimme sovelluksen kehitysympäristön perustamiseen ja sovelluksen ensimmäisten ominaisuuksiin liittyviä asioita seuraaavan listauksen mukaisesti.

1. Suunnitelma kehitysympäristöstä
2. Dokumentointi ja toimintatavat
3. XP -menetelmän soveltaminen
4. Sovelluksen toteutuksen ensimmäiset vaiheet
5. Sovitut toimenpiteet
6. Dokumentointi ja toimintatavat

Suunnitelma kehitysympäristöstä. Päätimme perustaa kehitysympäristö tehokkaalle kannettavalle tietokoneelle, johon asennetaan lähtötilanteessa työpöytäkäyttöön soveltuva Linux-käyttöjärjestelmä. Työasema sijoitetaan kollegan kotiverkkoon ja konfiguroidaan tarvittavien palveluiden osalta näkymään julkiseen verkkoon. Palvelin näkyy internetiin dynaamisella DNS-nimellä. Palveluna käytetään No-ip:tä (No-IP.com, 2017).

Sovelluksen tietokannaksi valittaan lähtötilanteessa PostgreSQL (The PostgreSQL Global Development Group, 2017), release 9.6.3. Mobiilisovellus toteutetaan Android -alustalle (Google, 2017). Kehitystyökaluna Android Studio ja ohjelmointikielenä Java.

Web-käyttöliittymän toteutetaan Python -kielellä (HTML5) (Python Software Foundation, 2017) ja Python kytkeytyy PostgreSQL -tietokantaan.

Dokumentointi ja toimintatavat. Projektin dokumentoinnissa käytetään Google Sheets ja Docs palveluja. Projektiin liittyvät tiedostot tallennetaan Google Driveen yhteiseen jaettuun “Team Drive” -hakemistoon. Projektissa tehdyt työt kirjataan ylös mahdollisimman tarkasti työmäärineen. Työmäärille tehdään Excel -taulukko, johon täydennetään todelliset työmäärät ja tehtyjen töiden selitykset. Projektin lähtötilanteessa ei käytetä yhteisesti mitään erillistä versionhallintaohjelmistoa. Projektin tekijät vastaavat itse oman osuuden varmistamisesta ja versionhallinnasta sopivaksi katsomallaan tavalla.

XP-menetelmän soveltaminen. XP-menetelmää noudatetaan soveltuvin osin. Projektin toteutus normaalin päivätyön lisäksi sekä pieni henkilöiden määrä aiheuttaa sen, että XP-menetelmää ei täydessä laajuudessaan voida soveltaa.

Sovelluksen toteutuksen ensimmäiset vaiheet. Ensimmäiset ominaisuudet sovellukseen toteutetaan alla lueteltujen käyttäjätarinoiden pohjalta.

- Sovellusten ulkoasun hahmotelmat esimmäisten toimintojen osalta.

- Konteista otetaan mobiilisovelluksella valokuvia, jotka tallentuvat tietokantaan.
- Tietokantaan mobiilisovelluksella tallennetut kuvat ovat tarkasteltavissa web-käyttöliittymästä.
- Tietokantaan tallennetaan valokuvan lisäksi kontin tunniste (ID-numero).

Ehdolla oli myös muita käyttäjätarinoita, joita ei toteuteta ainakaan ensimmäisessä vaiheessa. Tässä kohtaa toteuttamatta jätettävät käyttäjätarinat alla.

- Ominaisuus: valokuvan poistaminen tietokannasta mobiilisovelluksen avulla.
- Ominaisuus: valokuvan poistaminen tietokannasta web-käyttöliittymästä avulla.
- Käyttäjien todentaminen sovellukseen.

Sovitut toimenpiteet. Jaettiin ensimmäisen vaiheen työt tekijoiden kesken.

Minä:
- Tietokannan asennus palvelimelle.
- Mobiilikäyttöliittymän ensimmäisen suunnittelupelin toimintojen ohjelmointi.
- Sovelluksen ulkoasun luonnostelu.

Kollega:
- Palvelimen käyttöjärjestelmän asennus.
- Dynaaminen yhteysosoitteen hankkiminen palvelimeen.
- Tarvittavien palveluiden avaus palvelimelta julkiseen verkkoon.
- Web-käyttöliittymän ensimmäisen suunnittelupelin toimintojen ohjelmointi.
- Sovelluksen ulkoasun luonnostelu.

Ensimmäisen toteutusjakson pituus on 3 viikkoa (1.8.2017−22.8.2017).

4.4.2 Ensimmäisen toteutusjakson tulokset

Vaikka ensimäisen toteutusjakson työt aloitettiin välittömästi jakson alusta niin kaikkien sovittujen töiden tekeminen saatiin valmiiksi vasta toteutusjakson viimeisenä päivänä. Esimerkiksi palvelimen kytkeminen julkiseen verkkoon tietoturvallisesti vaati paljon enemmän työtä, kuin alunperin arvioimme. Lisäksi uusien ohjelmointiteknologioiden opiskeluun meni paljon aikaa. Kaikki sovitut työt saatiin kuitenkin tehtyä tavoitteiden mukaisesti.

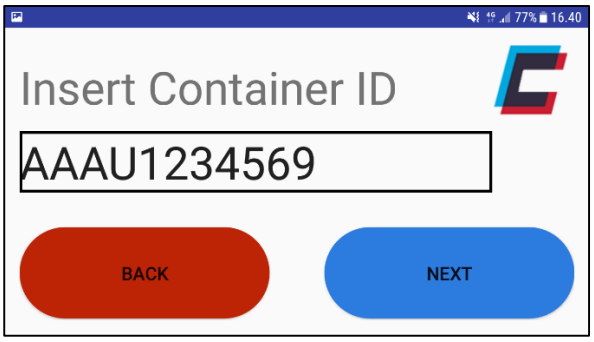
Suunnittelimme yhdessä mobiili- ja web-sovelluksen värimaailmaa ja ulkoasua, jotta ne olisivat mahdollisimman samanlaiset. Ohjelmoinnin aloittamisen yhteydessä Web-sovelluksen toteutus tarkentui tehtäväksi Django -sovelluskehyksellä (Django Software Foundation, 2017). Django on korkean tason Python Web -sovelluskehys jota on käytetty monen tunnetun internet-sivuston totetuksessa. Muunmuassa Instagram-, Twitter-, Pinterest- ja Nasa -sivustot käyttävät Django -sovelluskehystä.

Mobiilisovelluksen kehittämisessä ei tullut vastaan mitään suurempia ongelmia. Mobiililaitteena sovelluskehityksessä ja testauksessa käytetään Samsung Galaxy Xcover 4 (Android) -puhelinta ja mobiilisovellus optimoidaan muun muassa näytön koon osalta kyseisen puhelimen näytön kokoon (5.0 tuumaa). Koska mobiililaitteen näytön koko ei ole kovin suuri ja laitetta on tarkoitus käyttää ulko-olosuhteissa niin näyttöjen sisältö pidetään mahdollisimman yksinkertaisena.

Tietokantayhteys mobiilisovelluksesta tietokantaan toteutettiin suoraan sovelluksesta tietokantaan JDBC (Java Database Connectivity) -rajapinnan avulla. JDBC on Java-ohjelmointikielen rajapinta, joka määrittelee standardin tavan, millä asiakassovellus voi käyttää tietokantaa. Suora JDBC -rajapinnan käyttö vaatii sen, että mobiililaite on aina kytkeytyneenä mobiiliverkkoon, kun sovellusta käytetään. Jatkossa tietokantayhteys voidaan toteuttaa niin, että tietojen välivarastona mobiililaitteessa käytetään SQLite tietokantaa. Tällöin mobiililaitteen ei tarvitsisi aina olla kytkeytyneenä mobiiliverkkoon sovelluksen käytön aikana.

Kuva 11. *Container App -sovelluksen aloitusnäkymä.*

Aloitusnäkymästä siirrytään kontin tunnisteen syöttönäkymään *start*-painikkeella. sovellus suljetaan *exit*-painikkeella.



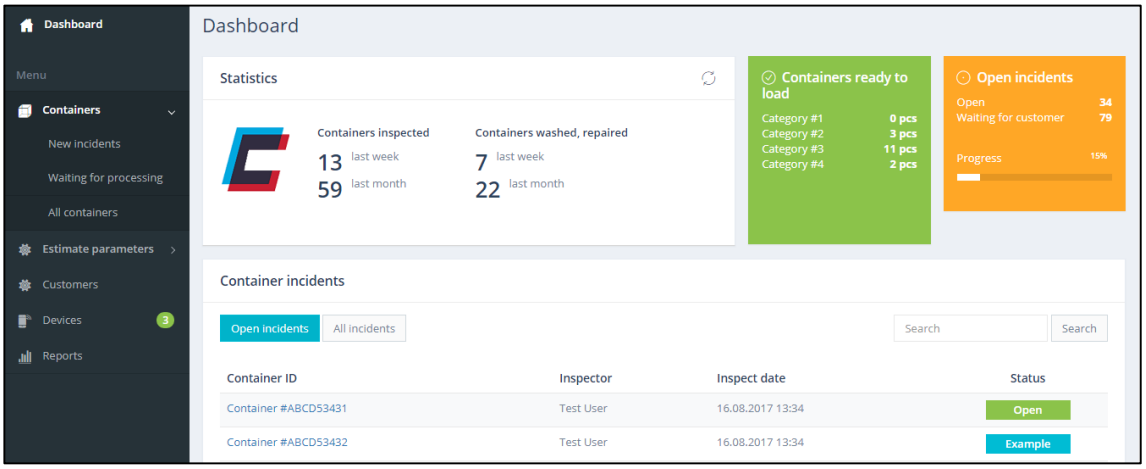
Kuva 12. *Kontin tunnisteen syöttö sovelluksessa.*

Kontin tunnisteen syöttönäytöllä syötetään kontin tunniste ja *next*-painikkeella siirrytään valokuvaustilaan. Aloitusnäytölle voidaan palata *back*-painikkeella.

Kuva 13. *Kontista otettu kuva Container App -sovelluksella.*

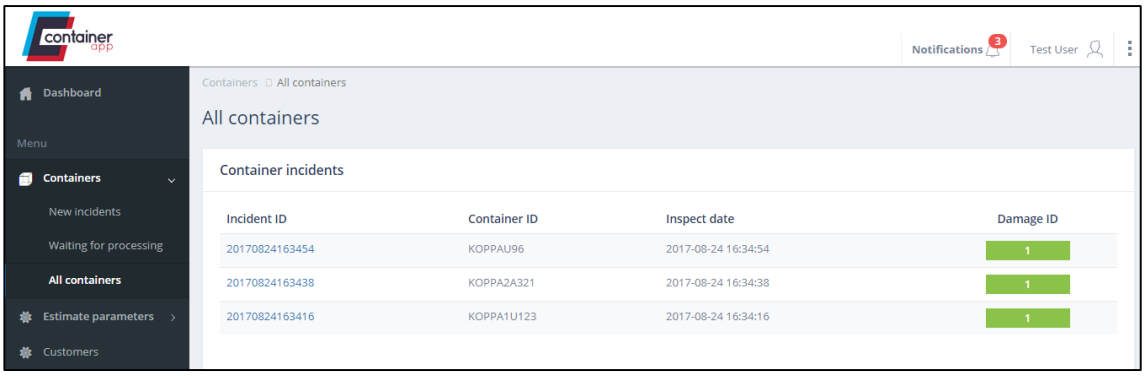
Valokuvan ottamisen jälkeen valokuva, kontin tunniste ja muut tarpeelliset tiedot talletetaan tietokantaan. Tietojen talletuksen jälkeen sovellus palaa kontin tunnisteen syöttönäytölle.

Web-sovelluksen sisällön suunnittelussa päätettiin, että alkuvaiheessa näytöille määritellään myös jonkin verran erilaisia näyttöelementtejä ja niissä esitetään jotain keksittyä kovakoodattua informaatiota. Tähän päädyttiin siksi, että työn toimeksiantajalle voidaan esitellä millaista erilaista informaatiota näytöillä olisi jatkossa mahdollista esittää.



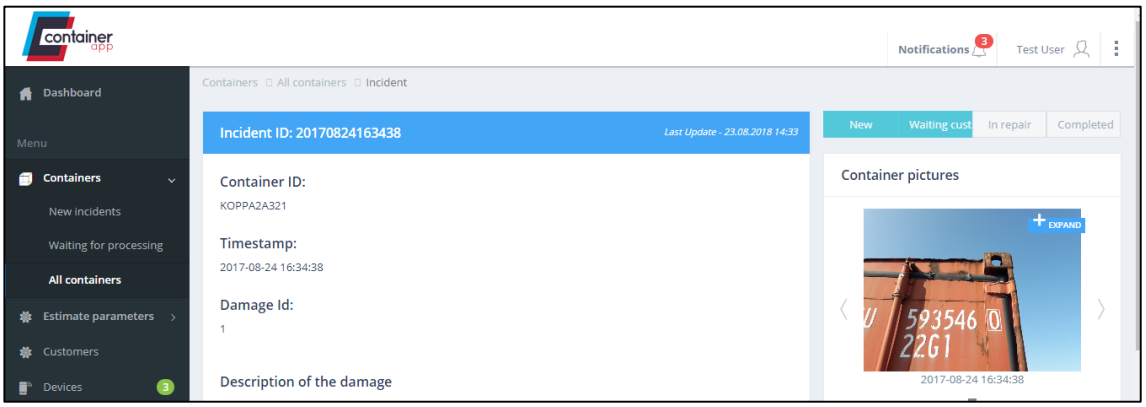
Kuva 14. *Web-sovelluksen aloitussivu.*

Django -sovelluskehyksellä saa tehtyä hyvin monipuolista ja näyttävää Web-käyttöliittymää. Kuvassa 14 on kuvaruutukaappaus web-sovelluksen aloitussivusta. Aloitussivulle on sijoiteltu erilaisia näyttöelementtejä.



Kuva 15. *Konttitapausten esitys.*

Rivin Incident ID -linkkiä klikkaamalla siirytään katsomaan tarkemmin konttiin liittyvää tapausta. Kuvassa 15 on rivilistaus tietokantaan talletetuista konttitapahtumista.



Kuva 16. *Konttitapahtuman tiedot.*

Kuvassa 16 on yksittäisen tietokantaan talletetun konttitapahtuman tietoja. Kontin kuva esitetään sivun oikeassa reunassa.

4.4.3 Ensimmäisen toteutusjakson retrospektiivi

Arvioimme 26.8.2017 ensimmäisen toteutusjakson onnistumisia ja haasteita. Omalta osaltani ensimmäisien toteutusjakson onnistumisia oli se, että mobiilisovelluksen käyttöliittymästä tuli selkeä ja se, että JDBC -rajapinnan avulla toteutettu tietokanta-yhteys toimii luotettavasti.

Kollega näki omalta osaltaa onnistumisena seuraavat asiat; Toteutusjaksoon määriteltyjen asioiden toteutus onnistui suunnitellussa aikataulussa vaikka tekemistä oli paljon. Lisäksi palvelimen perustaminen ja palvelun tarjoaminen tietoturvallisesti omasta verkosta vaati lisäopiskelua, mutta homma onnistui hyvin. Myös web-käyttöliittymästä tuli selkeä ja yhtenäinen kokonaisuus vaikka toteutukseen käytettiin uusia välineitä, joista oli etukäteen kokemusta vain vähän.

Lopputyön toimeksiantajalta tuli myös hyvää palautetta sovellukseen liittyen 25.8.2017 pitämässämme lopputyöpalaverissa. Palaverissa tuli useita kehitysehdotuksia jatkon toteutusjaksoille.

Vaikka ensimmäinen totetusjakso meni pääosin hyvin niin haasteitakin kuitenkin oli. Omalta osaltani mobiililaitteella otettujen kuvien skaalausta sopivaan kokoon ei saatu toteutusjakson puitteissa tehtyä. Kuvat ovat tarkkuudeltaan liian pieniä. Kuvien huono tarkkuus havaittiin vasta aivan toteutusjakson lopussa eikä aika riittänyt enää tilanteen korjaamiseen. Yleisesti ottaen Android -ohjelmoinnin mieleen palauttaminen vaati jonkin verran aikaa.

Kollegalla haasteina oli kuvatiedostojen esittäminen käyttöliittymässä. Kuvien vienti tietokannasta, oikeaan muotoon, palvelimen levylle oli hankala. Lopulta kuvat saatiin talentumaan palvelimen hakemistoon useaa eri toimintoa käyttämällä. Lisäksi ensimmäiseen toteutusjaksoon oli määritelty suhteellisen iso kokonaisuus ja uutta opeteltavaa oli paljon. Suuren työmäärän tekemiseen oli välillä haasteellista löytää aikaa.

4.5 Toinen toteutusjakso

4.5.1 Toinen suunnittelupeli

Toinen suunnittelupeli pidettiin 26.8.2017, jossa määrittelimme toisen toteutusjakson puitteissa toteutettavat työt. 25.8.2017 pidetyssä lopputyöpalaverissa laadimme työn toimeksiantajan edustajien kanssa joitain käyttäjätarinoita suunnittelupelissä käsiteltäväksi. Toisessa toteutusjaksossa toteutettavat sovelluksen ominaisuudet perustuvat seuraaviin käyttäjätarinoihin.

- Mobiilisovelluksella mahdollisuus ottaa yhteen konttitapahtumaan useampi kuva.
- Mobiilisovelluksella mahdollisuus määrittää kontin korjauksen/pesun tarve.
- Web-sovellukseen kontin omistajatieto näkyviin.
- Web-sovellukseen mahdollisuus selata kontteja omistajittain.
- Sovelluksen kehitys- ja demo -ympäristöjen eriyttäminen.
- Web-sovellukseen yksinkertainen estimointilaskenta.

Esillä oli myös muita käyttäjätarinoita joita ei toisen vaiheen toteutukseen valittu. Ainakin tässä kohtaa toteuttamatta jätettävät käyttäjätarinat olivat.

- Mobiilisovellukseen yksinkertaisempi näppäimistö.
- Mobiilisovellukseen puheohjattu konttin tunnisteen syöttö.

Myös ensimmäisen jakson tekemättä jätetyt käyttäjätarinat käytiin läpi mutta yksikään niistä ei noussut toteutettavien käyttäjätarinoiden listalle toiseen toteutusjaksoon.

Sovittiin toisen toteutusjakson työt tehtäväksi seuraavasti.

Minä:

- Sovittujen mobiilisovelluksen ominaisuuksien ohjelmointi.

Kollega:

- Sovittujen web-sovelluksen ominaisuuksien ohjelmointi.
- Sovelluksen kehitys- ja demo -ympäristöjen eriyttäminen.

Toisen toteutusjakson pituudeksi määritettiin kaksi viikkoa (30.8.2017−13.9.2017).

4.5.2 Toisen toteutusjakson tulokset

Toisen toteutusjakson kaikki tehtävät saatiin toteutettua aikataulussa. Heti toisen jakson aluksi sovelluksen kehitys- ja demoympäristöt eriytettiin. Tämä tehtiin sen vuoksi, että ensimmäisessä toteutusvaiheessa toteutettua sovellusta voidaan demonstroida Europortsilla vaikka uusien ominaisuuksien toteutus on käynnissä samaan aikaan.

Toisen toteutusjakson aikana sovelluksen kehitysympäristöön tehtiin myös fyysisiä muutoksia. Jo ensimmäisen toteutusjakson aikana havaitsimme, että kollegan laajakastayhteyden vaatimaton nopeus haittasi jossain määrin sovelluksen testausta ja kehityslaitteena toimiva työasema siirrettiin toisen kehitysjakson päätteeksi minun kotiverkkooni. Työaseman fyysisen siirron jälkeen muun muasssa kuvien esittäminen web-käyttöliittymässä nopeutui huomattavasti ja näin ollen myös sovelluksen testaaminen nopeutui hyvälle tasolle.

Uusien toimintojen ohjelmointi mobiili- ja web-sovellukseen vaati myös tietokannan taulurakenteen suunnittelua. Kollega suunnitteli tarvittavan tietokantarakenteen, joka sisältää kahdeksan taulua ja taulujen väliset yhteydet toteutettiin vierasavaimilla (Foreign Key (FK)). Toteutetut taulut:

- ca_customer (asiakkaan perustiedot),
- ca_container (kontin perustiedot),
- ca_customer_container (asiakkaan ja kontin välinen linkkitaulu),
- ca_incident (konttitapahtuman tiedot),
- ca_incident_img (konttitapahtuman kuvat),
- ca_incident_details (konttitapahtuman yksityiskohdat),
- ca_repair (korjaus-/pesutoimenpiteiden perustiedot) ja
- ca_repair_customer (asiakaskohtaisten korjaus-/pesutoimenpiteiden määritykset)

Mobiilisovellusta varten tietokantaan toteutettiin myös kaksi funktiorajapintaa tietojen tallettamista varten. Funktiorajapintojen tarkoituksena on pitää tietojen tallettaminen tietokantaan mahdollisimman yksinkertaisena mobiilisovelluksen puolella.

```
FUNCTION ca_ins_incident(in_incident_id   CHARACTER VARYING,
                         in_incident_desc CHARACTER VARYING,
                         in_container     CHARACTER VARYING)
```

Ohjelma 1. *Funktiorajapinta konttitapahtuman tallettamiseen.*

Funktiolle ca_ins_incident (ohjelma 1) annetaan parametrina konttitapahtuman (incident) identifioiva numero, konttitapahtuman kuvaus ja kontin tunnus. Konttitapahtuman kuvaus ei ole pakollinen tieto.

```
FUNCTION ca_ins_incident_img(in_incident_id     CHARACTER VARYING,
                             in_image_file      BYTEA,
                             in_image_filetype  CHARACTER VARYING,
                             in_repair_id       INTEGER)
```

Ohjelma 2. *Funktiorajapinta konttitapahtuman kuvan ja korjaus id:n talletukseen.*

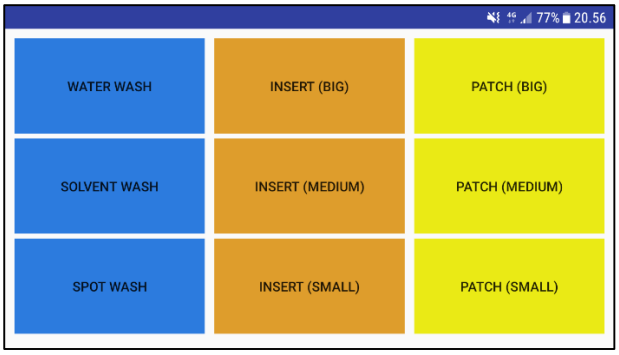
Funktiolle ca_ins_incident_img (ohjelma 2) annetaan parametrina konttitapahtuman (incident) identifioiva numero, konttitapahtuman valokuva bytearray -tietotyypiksi muutettuna, alkuperäisen valokuvan talletusformaatti sekä kontin korjauksen tai pesun korjauskoodi.

Samsung Galaxy Xcover 4 (Android) -puhelimella otetun valokuvan koko on noin 4–5 megatavun kokoinen ja valokuvaa voi hyvin skaalata jonkin verran pienemmäksi ilman, että sen laatu huononee liikaa. Valokuvan alkuperäinen koko on 4128 x 3096 pikseliä ja kokoa skaalataan mobiilisovelluksessa kokoon 1032 x 774 pikseliä ennen, kuin se muutetaan bytearray -tietotyypiksi ja talletetaan tietokantaan.

Toisen toteutusjakson tavoitteena oli muun muassa muuttaa mobiilisovellusta niin, että yhdelle konttitapahtumalle voidaan ottaa useampi kuva sekä mahdollistaa kontille korjauksen tai pesun tarpeen määritys. Kuvassa 17 näkyy mobiilisovelluksen näyttö jossa näytetään kontista otettu valokuva ja kontin vauriolle määritelty korjaustoimenpide. Kuvassa kontin vauriona on pahasti ruosteinen kohta joka pitää paikata. Paikkaus tehdään metallin palalla (patch), joka hitsataan vaurioitunen kohdan päälle. Kuvan korjaustoimenpide tässä esimerkissä on kuvitteelinen.

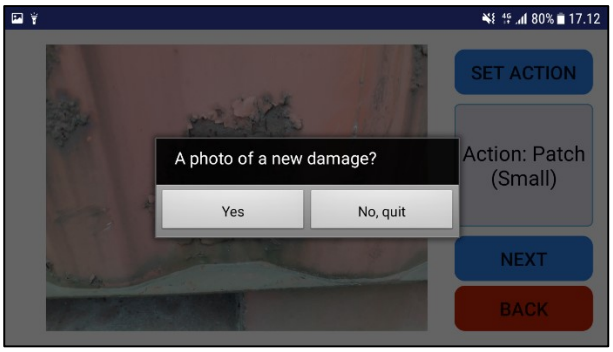
Kuva 17. *Valokuva kontin vauriosta ja korjaustoimepide asetettu.*

Kuvan 17 näytöltä voidaan siirtyä Korjaus- tai pesutoimenpiteen valinta -näytölle (kuva 18) *set action* -painikkeella. Korjaus- tai pesutoimenpiteen valinta -näytöllä on valittavissa esimerkinomaisesti yhdeksän eri toimenpidettä. Kolme pesutoimenpidettä ja kuusi paikkaustoimenpidettä. Korjaus- tai pesutoimenpide valitaan haluttua painiketta painikkeella.



Kuva 18. *Korjaus- tai pesutoimenpiteen valinta mobiilisovelluksessa.*

Kun haluttu korjaustoimenpide on valittu kuvan 18 näkymässä niin *next*-painikkeella siirrytään eteenpäin. Ohjelma voidaan sulkea *back*-painikkeella. Eteenpäin siirryttäessä ohjelma pyytää valitsemaan haluatko ottaa valokuvan toisesta kontin vauriosta samaan konttitapahtumaan (kuva 19). Jos näkymässä valitaan *yes*-painike niin avataan kamera uudelleen ja mahdollistetaan uuden kuvan ottaminen. Kuvan ottamisen jälkeen siirrytään taas kuvan 17 mukaiseen näkymään. Kun kaikki tarvittavat kuvat on otettu kyseiseen konttitapahtumaan niin kuvan 19 mukaisessa näytössä valitaan *no, quit* -painike joka tekee viimeisen talletuksen kyseiseen konttitapahtuman ja palaa sovelluksen aloitusnäytölle.



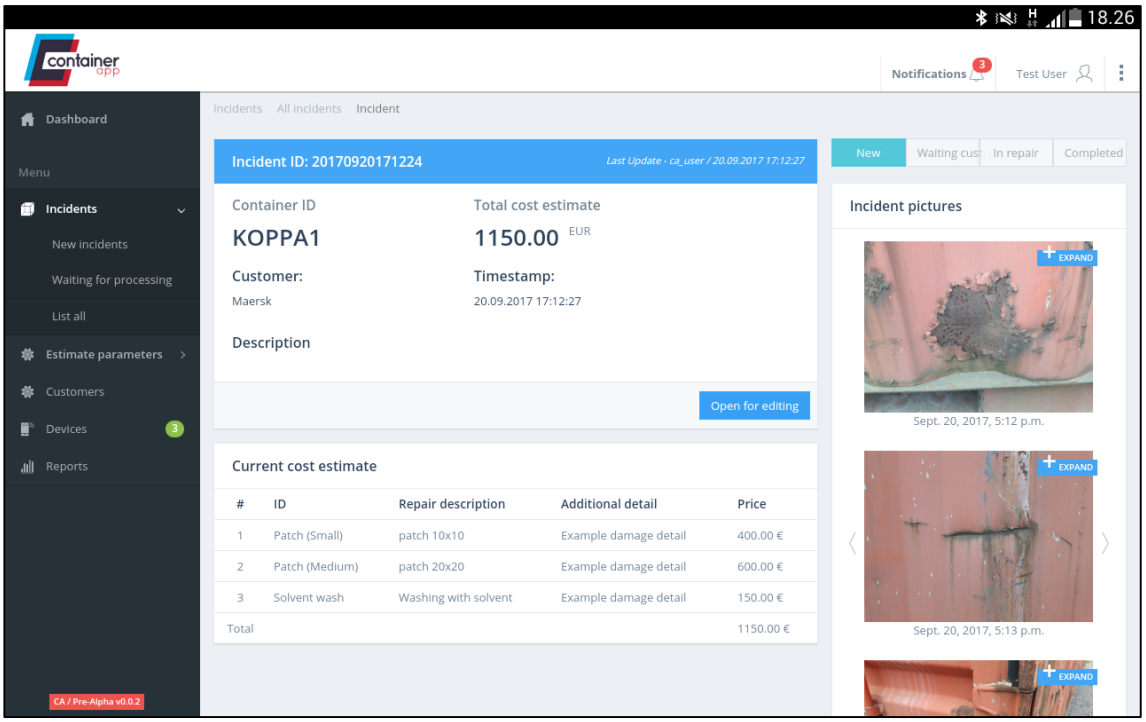
Kuva 19. *Otetaanko uusi valokuva vauriosta samaan konttitapahtumaan ?*

Toisen toteutusjakson tavoitteen web-sovelluksen puolella oli lisätä kontin omistajatieto näkyviin. Lisäksi web-sovelluksen näkymään haluttiin mahdollisuus selata kontteja omistajittain. Konttitapahtumien listausnäytölle (kuva 20) lisättiin mahdollisuus järjestää konttitapahtumia eri tiedon perusteella sekä rajata näytettävien tapahtumien määrää hakukentän (search) avulla.

Kuva 20. *Konttitapahtumien listauksessa järjestely- ja hakutoiminnot.*

Tavoitteena toisessa toteutusjaksossa oli myös toteuttaa sovellukseen yksinkertainen konttien vaurioiden arviointilaskenta. Arviointilaskenta on toteutettu niin, että mobiilisovelluksen puolella annettavalle korjaus- tai pesutomenpiteelle on määritelty tietokantaan hinta sekä yleisellä tasolla, että myös asiakaskohtaisesti. Kun tietokantaan talletetaan kontin vaurioden valokuvia ja korjaus- tai pesutoimenpiteitä, niin tiedot ovat nähtävissä web-sovelluksessa konttitapahtumakohtaisesti. Kuvan 21 näkymässä nähdään yksittäisen konttitapahtuman tiedot. Näkymässä keskellä ylhäällä nähdään muun muassa konttitapahtuman tunnistenumero (Incident ID), kontin tunniste (Container ID), asiakastieto (kontin omistaja) ja konttitapahtuman vaurioiden arviointilaskelma kokonaisuudessaan. Tässä kuvitteellisessa esimerkissä konttitapahtumassa määriteltiin kakasi eri korjaustarvetta ja yksi pesun tarve.

Kuvan 21 näkymän oikeassa reunassa näkyvät kontin vaurioista otetut valokuvat. Valokuvia pystyy suurentamaan internet-selaimen uuteen välilehteen, koko näytön kokoiseksi, kuvaa klikkaamalla.



Kuva 21. *Konttitapahtuman tiedot sisältävät valokuvat ja korjaushinta-arvion.*

4.5.3 Toisen toteutusjakson retrospektiivi

Toisen toteutusjakson onnistumisia ja haasteita arvioitiin 18.9.2017. Omalta osaltani arvioin toteutusjakson onnistumisiin muun muassa sen, että olen tyytyväinen moviilisovellukselle syntyneeseen ulkoasuun. Ulkoasu on hyvin selkeä ja yksinkertainen. Onnistumiseksi näen myös sen, että tietojen tallettaminen mobiilisovelluskesta tietokantaan on yksinkertainen kollegan tekeminen funktiorajapintojen ansiosta. Tietojen tallettamiseen tarkoitetut funktiorajapinnat vähentävät ohjelmointitarvetta mobiilisovelluksessa jonkin verran. Toisaalta sovelluksen suunnittelun yhtenä pääajatuksena on ollut se, että mahdollisimman paljon sovelluslogiikkaa ohjelmoidaan tietokantaan. Tällainen toimintatapa mahdollistaa helpommin erilaisten päätelaitteiden integroinnin sovellukseen.

Kollegan mielestä toisen toteutusjakson onnistumisiin kuuluu hyvän tietokantamallin valitseminen ja onnnistunut toteuttaminen heti ensimmäisellä yrittämällä. Kun tietokantamallin taulut, funktiot ja liipaisimet (trigger) oli kertaalleen tehty niin niitä ei enää tarvinnut enää muuttaa kehitysjakson aikana. Kollega oli tyytyväinen myös siihen, että web-käyttöliittymän muutokset onnistuivat suoraviivaisesti, koska ohjelmointi-työkalut olivat tulleet jo tutummaksi. Jonkin verran web-sovelluksen rakennetta kirjoitettiin myös uudelleen (refaktorointi), kun ensimmäisessä toteutusvaiheessa toteutettua web-sovellusta käytettiin pohjana uudelle kehitysympäristölle.

Haasteina toisen toteutusjakson alussa oli mobiililaitteen kameralla otetun kuvan skaalaaminen pienemmäksi. Tämä ongelma havaittiin jo ensimmäisen toteutusjakson lopulla. Sopivan toimintatavan löytäminen vai aikaa ja vaati paljon testausta. Valokuvan skaalaaminen sopivaan kokoon onnistui kuitenkin lopulta hyvin. Lisäksi taaksepäin liikkuminen mobiilisovelluksessa aiheutti jonkin verran haasteita. Oikeaan edelliseen näkymään löytämien ei ollut aivan yksinkertaista ja toimintalogiikan ohjelmointi vaati aika paljon enemmän työpanosta, kuin olin alunperin arvioinut. Myös useamman valokuvan ottaminen yhteen konttitapahtumaan aiheutti aluksi vaikeuksia, koska mobiililaitteen muisti loppui kesken jo muutaman otetun valokuvan jälkeen. Tämä korjaantui lopulta mobiilisovelluksen toimintalogiikan muutoksella.

Kollegalla toisen toteutusjakson suurimmat haasteet tulivat web-käyttöliittymän eriyttämisessä kehitys- ja demoympäristöön. Toimenpide aiheutti paljon selvittämistä ja toteutustyötä. Ensimmäisellä yrityksellä valittu ympäristöjen eriyttämistapa osoittautui käyttökelvottomaksi ja tämän johdosta piti selvittää toinen tapa. Oikean toimintatavan löydyttyä kehitys- ja demoympäristöt toimivat moitteettomasti.

Kaiken kaikkiaan toisesta toteutusjaksosta jäi erittäin positiiviset muistot. Sekä web- että mobiilisovelluksen toimintalogiikka monimutkaistui huomattavasti ensimmäiseen toteutusjaksiin nähden mutta sovellusten toimintavarmuus pysyi kuitenkin hyvänä.

4.6 Kolmas toteutusjakso

4.6.1 Kolmas suunnittelupeli

Kolmas ja viimeinen suunnittelupeli järjestettiin 26.9.2017. Kolmas suunnittelupeli ja kolmas toteutusjakso päättävät diplomityöprojektini ja tämän johdosta mietimme työn toimeksiantajan sekä kollegani kanssa hyvin tarkkaan mitä Container App -sovellukseen tässä kohtaa vielä toteutetaan. Kolmannessa suunnittelupelissä käyttäjätarinoita tuli useita ja edellisissä suunnittelupeleissä toteuttamatta jätetyt käyttäjätarinat käytiin myös läpi. Kolmanteen toteutusjaksoon tuli valituksi seuraavat käyttäjätarinat:

- Mahdollisuus käyttää mobiilisovellusta vaikka langatonta- tai mobiiliverkkoa ei ole saatavilla.
- Kuvauksen lisääminen konttitapahtumaan web-sovelluksessa.

Esillä oli myös muita käyttäjätarinoita joita ei kolmannen vaiheen toteutukseen valittu. Toteuttamatta jätettävät käyttäjätarinat ovat:

- Useamman korjauksen/pesun tarpeen lisääminen yhteen valokuvaan.
- Useamman valokuvan ottaminen yhteen korjauksen/pesun tarpeeseen.
- Kommenttien lisääminen konttitapahtumaan mobiilisovelluksessa.

Sovittiin kolmannen toteutusjakson työt tehtäväksi seuraavasti.

Minä:

- Mobiilisovelluksen ominaisuuksien ohjelmointi niin, että mobiilisovellusta on mahdollista käyttää vaikka langatonta- tai mobiiliverkkoa ei ole saatavilla.

Kollega:

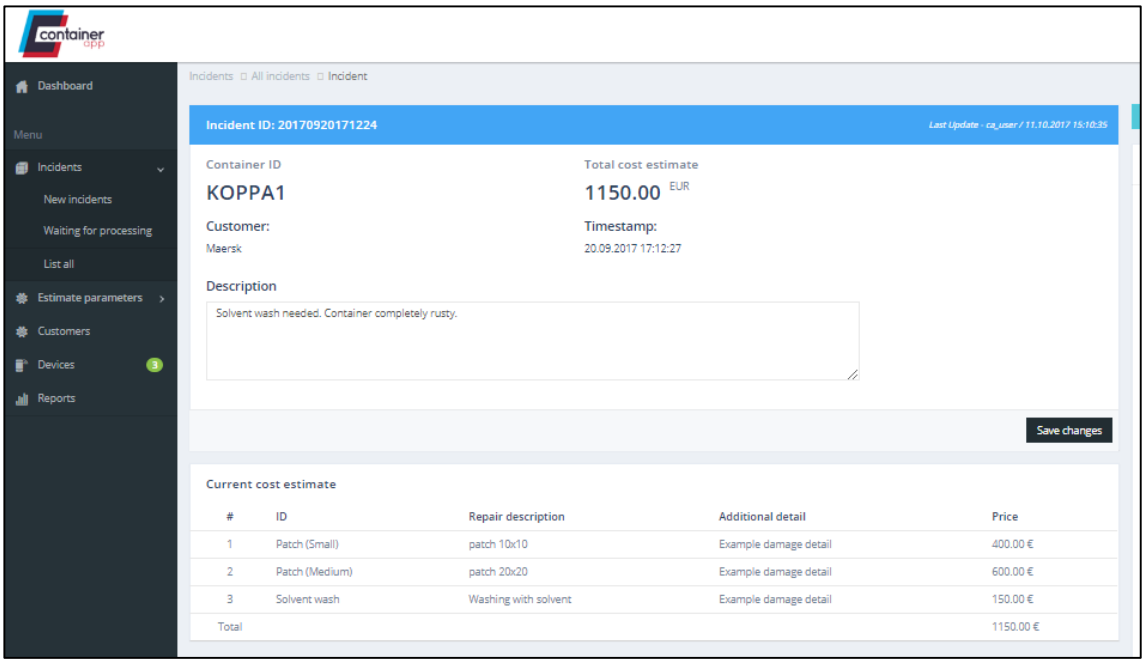
- Sovittujen web-sovelluksen ominaisuuksien ohjelmointi.
- Palvelinpään ohjelmointi niin, että mobiilisovelluksen käyttö onnistuu vaikka langatonta- tai mobiiliverkkoa ei ole saatavilla.

Käyttäjätarinoita ei kolmanteen toteutusjaksoon valittu, kuin kaksi mutta arvioimme, että sovelluksen toiminnan muuttaminen sellaiseksi, että mobiilisovellus voi toimia ilman langatonta- tai mobiiliverkkoa tulee vaatimaan reilusti työtä.

Kolmannen toteutusjakson pituudeksi määritettiin kaksi viikkoa (28.9.2017–12.10.2017).

4.6.2 Kolmannen toteutusjakson tulokset

Myös kolmanteen toteutusjaksoon suunnitellut työt saatiin tehtyä aikataulussa. Kolmannessa toteutusjaksossa toteutetut ominaisuudet näkyvät ulkoisesti vaan web-sovelluksessa jossa toteutettiin kuvauksen lisäysmahdollisuus konttitapahtumaan. Kuvauksen lisäys esitetty kuvassa 22.



Kuva 22. *Kuvauksen lisääminen konttitapahtumalle Description -kenttään.*

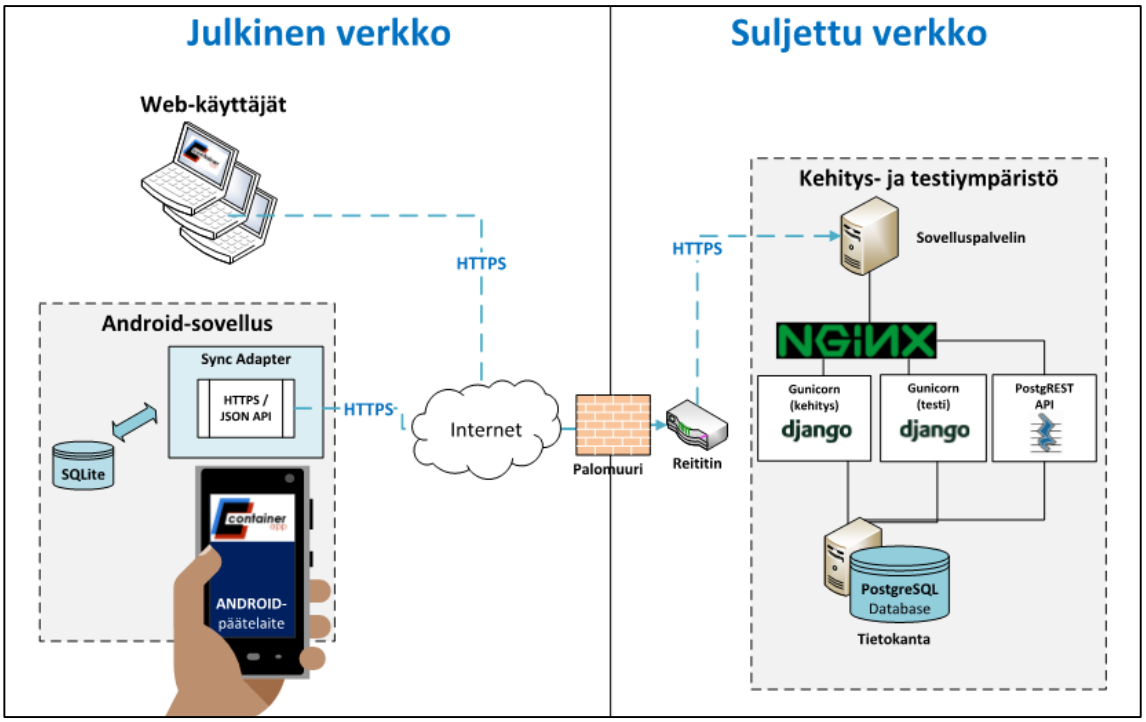
Suurimmaksi osaksi toteutusjaksossa keskityttiin mobiilisovelluksen ja palvelinohjelmiston ominaisuuksien muuttamiseen niin, että mobiilisovellusta on mahdollista käyttää vaikka langatonta- tai mobiiliverkkoa ei ole saatavilla. Mobiilisovelluksen osalta tarvittavia ominaisuuksia alettiin ohjelmoida Androidin sisäänrakennetun Sync Adapter -rakenteen (Android.com, 2017) ympärille ja palvelinohjelmistoa alettiin toteuttaa palvelimelle asennetun PostgREST web-palvelimen (Nelson, 2017) avulla.

Ensimmäisessä toteutusjaksossa mobiilisovelluksen tietokantayhteyden toteutukseen valittiin JDBC-rajapinta joka on huomattavasti helpompi toteuttaa yksinkertaiseen sovellukseen, kuin Sync Adapter -rakenteen käyttö. Heti ensimmäisen toteutusjakson alussa tutustuin jonkin verran Sync Adapter:iin mutta päätin jättää sen tarkemmin tutkittavaksi myöhemmässä vaiheessa.

Yksinkertaisesti selitettynä mobiilisovellusta muutettiin niin, että siihen lisättiin oma sisäinen tietokanta, jota synkronoidaan palvelimella olevan PostgresSQL-tietokannan (The PostgreSQL Global Development Group, 2017) kanssa. Jos mobiililaite on kytkeytyneenä langattomaan- tai mobiiliverkkoon niin konttitapahtuman tiedot siirtyvät mobiililaitteen sisäisestä tietokannasta palvelimen tietokantaan välittömästi. Jos taas mobiililaite ei syystä tai toisesta ole kytkeytyneenä verkkoon niin tiedot synkronoidaan sisäisestä tietokannasta palvelimen tietokantaan, kun verkkoyhteys saadaan taas muodostettua. Sync Adapter -rakenne huolehtii muun muassa synkronoinnin automaattisen suorittamisen, verkkoyhteyksien olemassa olon tutkimisen, käyttäjätilien hallinnoimisen ja tunnistautumisen palvelimelle.

Mobiililaite muodostaa suojatun https-yhteyden sovelluspalvelimen NGINX-http-palvelimeen, jonka kautta tarjotaan JSON-rajapintaa (PostgREST). Rajapinta palauttaa pyynnöistä paluukuittauksen mobiililaitteelle JSON-muodossa. Pyynnön mukaisen toimenpiteen suorittaa lopulta tietokannassa olevaan ohjelmayksikkö, joka lisää tai muuttaa tietoa tietokannan tauluissa.

Tietokantataulujen osalta päätettiin, että tietojen muokkaamista ei sallita suoraan JSON-rajapintaa käyttämällä, vaan tietokantamuutokset tehdään tietokantaan toteutettujen rajapintaohjelmien kautta. Kuvatiedostot muunnetaan mobiililaitteella BASE64-muotoon, jotta niitä on mahdollista siirtää JSON rajapintaa käyttämällä. Kuvassa 23 on kuvattu Container App -sovellukselle syntynyt arkkitehtuuri.



Kuva 23. *Container App -sovelluksen arkkitehtuuri.*

4.6.3 Kolmannen toteutusjakson retrospektiivi

Kolmannen ja viimeisen toteutusjakson onnistumisia ja haasteita arvioitiin 12.10.2017. Arvioin omiin onnistumisiini sen, että sain Androidin Sync Adapter -rakenteen toimimaan luotettavasti. Omalta osaltani tämä toteutusjakso vaati eniten työtunteja ohjelmoinnin osalta. Toteutusjaksossa kokeilin useamman eri ohjeen perusteella ominaisuuksien ohjelmointia ennen, kuin löysin hyvän lopputuloksen toteutukselle.

Kollega nosti kolmannnen toteutusjakson onnistumisiin muunmuassa sen, että asennukseen ja konfiguroitiin liittyvien vaikeuksien jälkeen REST-rajapinta toimi nopeasti ja luotettavasti. Lisäksi JSON -rajapinnan käyttö uudella tavalla oli menestys ja tietokantaan perustettuja ohjelmia oli mahdollista käyttää uuden rajapinnan avulla.

Haasteita toteutusjaksossa oli riittävästi. Sync Adapter -rakenteen ja JSON-tiedostomuodolla toteutetun tietojen siirron lisääminen mobiilisovellukseen lisäsivät sovelluksen monimutkaisuutta huomattavasti aikaisemmasta ja erilaisia haasteita ilmeni. Valokuvat aiheuttivat taas ongelmia ja jouduin pakkaamaan kuvia jonkin verran, jotta palvelinohjelmisto otti kuvia vastaan sujuvasti. Alkuperäisen kokoiset valokuvat olivat ilmeisesti kooltaa liian isoja ja PostgREST web-palvelin tukkeutui kuvista. Päätimme kollegan kanssa, että kuvien kokoa pienennetään, jotta rajapinta saadaan toimimaan luotettavasti. Kuvien tarkkuus ei juurikaan pakkauksesta kärsinyt.

Kollegalla haasteina tässä toteutusjaksossa oli se, että REST-rajapinnan mahdollistavan PostgREST-rajapinnan konfigurointi piti tehdä useaan kertaan, koska kaikkia

toimenpiteitä ei ollut täysin dokumentoitu manuaaliin. Haasteita tuotti myös se, että kaikkia palvelimelle asennettavia ohjelmia ei ollut käännettynä palvelimen käyttöjärjestelmälle. Tämän johdosta piti opetella ohjelmien kääntäminen Linux-käyttöjärjestelmässä.

4.7 Europortsin kommentit sovelluksesta

Container App -sovellus oli esiteltävänä ja testattavana Europortsilla jo toisen toteutusvaiheen jälkeen, koska mobiili- ja web-käyttöliittymä oli siinä vaiheessa toteutettu jo melkein loppulliseen muotoonsa. Sovellusta oli esitelty Europortsilla joillekin yksittäisille kontin tarkastajille ja yleinen vastaanotto oli ollut positiivista. Odotettavissa oli jo etukäteen, että sovelluksen keskeneräisyys nousee esille, kun sovellusta esitellään potentiaalisille loppukäyttäjille. Palautteita ja kehitysehdotuksia on listattu alla.

Selkeästi positiivinen palaute:

- Sovellus on selkeä käyttää.
- Rakenne näyttää suhteellisen loogiselta tähän mennessä toteutetuilla toiminnoilla.
- Sovellus toimii kohtuullisen nopeasti tällä rakenteella ja tietomäärällä.

Kehitysehdotuksia ja muita huomioita:

- Kaikki testauksen yhteydessä tehdyt konttitapahtumat eivät tallettuneet tietokantaan.
- Konttien tiedot olisi hyvä olla mobiililaitteessa ladattuna, jotta ennakoiva konttitunnuksen syöttö olisi mahdollista.
- Mobiilisovelluksen käytettävyys talviolosuhteissa, hanskat kädessä, arvelutti. Näytölle sopiva kynä voisi olla ratkaisu tähän.
- Mobiilisovelluksessa olisi myös hyvä pystyä antamaan omia kommentteja. Kommentit voisivat olla sekä vapaatekstiä, että ennakkovalintaisia huomautuksia.

Yleisenä kommenttina oli, että sovelluksessa olisi potentiaalia konttien tarkastusta helpottavaan ja nopeuttavaan ratkaisuun. Itse sovellus vaatisi kuitenkin vielä melko paljon jatkokehittelyä, jotta kaikki tarvittava toiminnallisuus saataisiin siihen toteutettua.

5. POHDINTA JA ANALYYSI

5.1 Lopputuloksen arvionti

Diplomityössäni tutkittiin tietojärjestelmän arkkitehtuurin syntymistä ketterässä ohjelmistokehityksessä. Diplomityön puitteissa toteutettiin merikonttien kunnon ja laadun valvontaan tarkoitettua tietojärjestelmää jolla arkkitehtuurin syntymistä voitiin arvioida.

Ketterässä ohjelmistokehityksessä periaatteena on suunnitella kehitystä, tietyllä ajanhetkellä, vain sen verran eteenpäin, että seuraavan toteutusjakson työt saadaan tehtyä. Tämä ajatus- ja toimintamalli ei kuitenkaan pysynyt itselläni jatkuvasti mielessä ja ajatukset toteutuksesta karkailivat pidemmälle jo mahdollisesti useamman toteutusjakson yli. Esimerkiksi ensimmäisessä toteutusjaksossa valittu tietokantayhteyden toteutus JDBC-rajapinnan avulla oli selkeästi väliaikainen ratkaisu joka tehtiin vaan siksi, että toimivaa sovellusta saatiin nopeammin esitettyä työn toimeksiantajalle. Todellisuudessa sovelluksen toimiminen oikeassa toimintaympäristössä vaatii sen, että mobiilisovellusta voi käyttää suunniteltuun käyttöön myös silloin, kuin langatonta- tai mobiiliverkko ei ole käytettävissä.

Ei ole epäilystäkään siitä, etteikö ketterä ohjelmistokehitysmenetelmä ollut paras vaihtoehto sovellusprojektin toteuttamiseen. On todella vaikeaa kuvitella, että kaikki Container App -sovellukseen toteutetut ominaisuudet, valitut teknologiat ja välineet olisi pitänyt määritellä etukäteen arkkitehtuurilähtöisen ohjelmistokehitysmenetelmän mukaisesti. Diplomityön laajuus asettaa rajat sille mitä diplomityön puitteissa pystytään toteuttamaan ja lisäksi etukäteen ei pystynyt tekemään kovin tarkkoja arvioita siitä mitä työn toimeksiantaja haluaisi sovellukseen toteutettavaksi. Etukäteen ei myöskään voinut tietää millaisen työmäärän joku työn toimeksiantaja haluama sovelluksen ominaisuus vaatisi.

Ketterä ohjelmistokehitysmenetelmä sopiii erittäin hyvin pienessä ryhmässä totetettavaan ohjelmistoprojektiin. Varmasti menetelmiin liittyvät haasteet lisääntyvät, kun tekijöiden määrä projektissa lisääntyy. Ohjelmiston arkkitehtuurista syntyy myös dokumentaatiota suunnittelupelien ja retrospektiivien dokumettien pohjalta. Suunnittelupelien yhteydessä laadittut kuvat (esimerkiksi kuva 23) ja hahmotelma sovelluksen toteutuksesta toimivat myös tärkeänä dokumentaationa, kun esimerkiksi joskus myöhemmin sovelluksen toimitaperiaatteita selvitetään.

5.2 Tulokset

Vaikka Container App -sovelluksen karkean tason arkkitehtuuri oli nähtävissä jo siinä vaiheessa, kun sovellusta aluksi hahmoteltiin niin sovelluksen useiden eri ominaisuuksien toteutustapa muuttui kehityksen edetessä. Aluksi lähdettiin tietoisesti siitä, että haluttiin kohtalaisen nopeasti esittää työn toimeksiantajalle jotain valmista helppokäyttöistä mobiilisovellusta ja näyttävää web-käyttöliittymää. Sovelluksen toiminnallisuutta oli sitten suunnitelmissa toteuttaa työn toimeksiantajan sekä minun ja kollegani ajatusten pohjalta.

Tietokantayhteyden muodostus ja tietokannan rakenne sekä tietojen siirto mobiililaitteesta palvelimen tietokantaan olivat sellaisia toimintoja joiden toteutustavat muuttuivat sovelluskehityksen eri toteutusjaksossa. Muun muassa näiden toimintojen muuttuminen sovelluskehityksen edetessä kuvastaa sitä miten sovelluksen arkkitehtuuri muodostuu sovelluksen sisäisten ja työn toimeksiantajan tarpeiden perusteella. Näitä tarkastellaan seuraavissa aliluvuissa tarkemmin.

5.2.1 Tietokantayhteys ja tietokannan rakenne

Aluksi sovellukseen haluttiin hyvin yksinkertainen rakenne, jottei sen suunnitteluun ja toteutukseen kuluisi mahdottomasti aikaa. Ensimmäisessä toteutusjaksossa sovelluksessa oli vain yksi taulu johon talletettiin mobiililaitteelta konttitapahtuman tunnus, kontin tunnus ja valokuva. Tietokantayhteys toteutettiin JDBC-rajapinnan avulla.

Toisessa toteutusjaksossa mobiilisovellukseen totetettiin mahdollisuus ottaa yhteen konttitapahtumaan useampi kuva. Lisäksi toteutettiin kuvan oton yhteydessä mahdollisuus määrittää kontin korjauksen tai pesun tarve. Tässä kohtaa päätettiin myös pohdiskella tietokannan taulurakennetta tarkemmin ja rakentaa se heti kerralla kunnolla. Tietokantatauluja tuli sovellukseen seitsemän lisää. Tässä toteutusjaksossa tietokantayhteys toteutettiin edelleen JDBC-rajapinnan avulla

Kolmannessa toteutusjaksossa mobiilisovellus haluttiin muuttaa toimimaan niin, että se toimii myös silloin, kun laite ei ole kytkeytyneenä langattomaan- tai mobiiliverkkoon. Jos sovellusta pitää pystyä käyttämään myös silloin, kun mobiililaite ei ole kytkeytyneenä verkkoon niin laitteessa pitää olla oma tietokanta jonne tiedot puskuroidaan ja viedään varsinaiseen tietokantaan sieltä, kun laite kytkeytyy verkkoon. Mobiilisovellusta muutettiin niin, että sovelluksen tallentamat tiedot talletetaan sovelluksen sisäiseen tietokantaan. Tietojen siirto mobiililaitteen tietokannasta varsinaiseen tietokantaan toteutettiin https-yhteydellä sovelluspalvelimen NGINX-http-palvelimeen JSON-tiedostomuodolla. JDBC-rajapinnasta siis luovuttiin tässä vaiheessa. Varsinaisen tietokannan rakenteeseen ei tässä toteutusjaksossa enää tullut muutoksia.

5.2.2 Tietojen siirto palvelimelle

Ensimmäisessä toteutusjaksossa Container App -sovelluksen tiedot talletettiin suoralla SQL-lauseella varsinaiseen tietokantaan palvelimelle. Tietokantatauluja sovelluksessa oli tässä vaiheessa ainoastaan yksi ja tietojen talletus oli hyvin yksinkertainen operaatio.

Toisessa totetutusjaksossa tietokantaan tuli lisää tauluja ja taulujen rakenne muuttui sellaiseksi, että tietojen tallennus suoralla SQL-lauseella tauluihin olisi ollut hankalaa. Tämän johdosta tietokantaan toteutettiin kaksi funktiorajapintaa joiden avulla tiedot talletettaan tietokantatauluihin. Funktiorajapintojen avulla mobiilisovelluksessa ei tarvitse tietää tietokannan taulurakenteesta mitään. Tietokantafunktioille annetaan vaan sen pyytämät tiedot ja se huolehti tietojen talletuksesta tietokantatauluihin.

Kolmannessa toteutusjaksossa suorasta yhteydestä varsinaiseen tietokantaan luovuttiin ja siirryttiin tallettamaan tiedot mobiililaitteen sisäiseen tietokantaan. Sisäisestä tietokannasta tiedot viedään JSON-tiedostomuodossa sovelluspalvelimen http-palvelimeen. Pyynnön mukaisen toimenpiteen suorittaa lopulta tietokannassa olevaan tietokantafunktio, joka lisää tai muuttaa tietoa tietokannan tauluissa.

5.3 Pohdinta

Beck (1999) selittää arkkitehtuurin syntymisen Extreme Programming -menetelmässä niin, että ensimmäiseen iteraatioon valitaan joukko yksinkertaisia perustarinoita joiden odotetaan pakottavan luomaan koko arkkitehtuurin. Tämän jälkeen kavennetaan horisonttia ja totetetaan tarinat mahdollisimman yksinkertaisella tavalla. Kun nämä toimenpiteet on tehty niin lopuksi arkkitehtuuri on syntynyt.

Diplomityössä päätutkimuskysymyksen aiheeksi asetettiin merikonttien kunnon ja laadun valvontaan tarkoitetun tietojärjestelmän arkkitehtuurin syntyminen ketterässä ohjelmistokehityksessä. Toteutetun Container App -sovelluksen toteutuksen edetessä pystyi selvästi havaitsemaan sovelluksen arkkitehtuurin syntymisen. Arkkitehtuuri täydentyi ja osin myös muuttui ohjelmistoprojektini kolmessa toteutusjaksossa.

Diplomityöhöni liittyvässä ohjelmistoprojektissa ei tehty raskasta suunnitteluprosessia aluksi kuten arkkitehtuurilähtöisessä ohjelmistokehityksessä tehdään. Arkkitehtuurilähtöisessä mallissa laaditaan säännöt, joita tietyn arkkitehtuurin mukaan rakennettavassa järjestelmässä on noudatettava, voivat koskea teknologian käyttöä, algoritmien valintaa, tietorakenneratkaisuja, sekä suunnittelu- ja toteutusmalleja. Koskimiehen ja Mikkosen (2005) mukaan voidaan ajatella, että arkkitehtuuri on järjestelmän peruslaki, jota voidaan muuttaa vain erittäin painavilla perusteilla. En usko, että alussa tehdyn tiukan suunnitteluprosessin pohjalta olisi syntynyt yhtä hyvää ratkaisua, kuin mitä nyt syntyi. Sovelluksen arkkitehtuuri muotoutui kuitenkin jonkin verran erilaiseksi, kuin olimme alunperin suunnitellut.

Ketterä ohjelmistokehitys oli siis oikea valinta ohjelmistokehityksen menetelmäksi. Leffingwellin (2007) mukaan ketterässä ohjelmistotuotannossa ei tehdä raskasta suunnitteluprosessia aluksi, vaan järjestelmän rakenne muodostuu asiakkaan ja kehitysryhmän valitsemien toteutettavien ominaisuuksien mukaan. Ohjelmiston arkkitehtuuri muodostuu inkrementaalisesti kehitysjaksosta toiseen. Ketterä ohjelmistokehitysmenetelmä sopi myös siksi erittäin hyvin ohjelmistoprojektiini, koska minulla ei ollut kovin hyvää kuva siitä miten paljon sovellusta kerittäsiin kehittää. Jos sovellusta olisi lähdetty toteuttamaan tiukan etukäteissuunnitelman pohjalta niin olisi ollut vaarana, että jokin sovelluksen hankalasti toteutettava ominaisuus olisi vienyt paljon aikaa heti alussa ja näin ollen se olisi aiheuttanut hankaluuksia kehityksen etenemisessä. Pahimmassa tapauksessa aika olisi loppunut kesken ja emme olisi saaneet mitään esiteltävää työn toimeksiantajalle.

Se miten Beckin selitys määrittelee ohjelmistoarkkitehtuurin toimii mielestäni hyvin karkealla tasolla. Aluksi pitää tietenkin olla jotain näkemystä siitä miten sovellus voitaisiin toteuttaa. Omaan diplomityöhöni liittyvässä ohjelmistoprojektissa tiedettiin alussa, että tarvitaan mobiililaite, jolla kuvataan merikonttien vaurioita. Ei ollut tiedossa olisiko mobiililaite mahdollisesti taulutietokone vai puhelin. Ei myöskään oltu päätetty olisiko mobiililaitteen käyttöjärjestelmä Android, iOS vai Windows. Nähtäväksi jäisi esimerkiksi se, että soveltuuko puhelimen 5-tuumainen näyttö tarvittavien tietojen näyttämiseen ja syöttämiseen. Lisäksi käyttöolosuhteiden valoisuuden ja sääolojen vaikutus mobiililaitteen käytettävyyteen oli täysin arvoitus ja mobiililaitteen pitäisi kuitenkin toimia kaikissa sääoloissa. Mietin myös jossain kohtaa, että miten pitkä mobiililaitteen käyttöikä on kovissa olosuhteissa. Mobiililaitteen kestävyyttä vaativissa toimintaympäristön olosuhteissa ei tämän diplomityön puittessa kuitenkaan koitettu selvittää.

Ohjelmistoprojektin alussa tiedettiin myös se, että tarvitaan tietokanta kuvien ja muiden tietojen tallettamiseen. Oli selvää, että tietokannan pitää olla jossain ulkoisella palvelimella, koska ei tietoja pysty selailemaan laitteen paikallisesta mobiililaitteen tietokannasta, kuin paikallisesti siinä kyseisessä mobiililaitteessa jossa tiedot ovat talletettuna. Tietokantaratkaisuksi oli tarjolla eri vaihtoehtoja. Vaihtoehtoina tietokannoiksi oli muunmuassa perinteinen relaatiotietokanta ja esimerkiksi NoSQL-tietokanta. Projektin alussa oli siis paljon avoimia kysymyksiä mutta moniin niistä löydettiin nopeasti vastaus ja näin ohjelmiston arkkitehtuuri alkoi muodostua.

Vaikka suurimman osan ketteristä ohjelmistokehitysmenetelmistä sanotaan kiinnittävän hyvin vähän huomiota arkkitehtuurin suunnnitteluun niin Extreme programming (XP) -menetelmän kehittäjä Beck (1999) on kuitenkin sitä mieltä, että arkkitehtuuri on yhtä tärkeä XP-projektissa, kuin missä tahansa muussa ohjelmistoprojektissa. Vaikka XP-menetelmässä yksikään käytännöistä ei nimenomaisesti osoita ohjelmistoarkkitehtuurin roolia niin tämä ei kuitenkaan tarkoita, etteikö XP-menetelmä tai -tiimi ymmärtäisi arkkitehtuurin roolia sovelluskehityksessä. Beckin mukaan arkkitehtuuri voi ilmetä

päivittäisessä suunnittelussa ja se syntyy koodista pikemminkin, kuin minkäänlaisena etukäteisuunnitelmana. Karkealla tasolla Container App -sovelluksen arkkitehtuuri syntyi ensimmäisen iteraation perustarinoista ja hienomman tason arkkitehtuuri muodostui hyvin pitkälti päivittäisten valintojen perusteella. Päivittäiset valinnat taas syntyivät siitä miten sovelluksen toimintoja oli järkevintä ohjelmoida. Voidaan siis todeta, että Container App -sovelluksen arkkitehtuuri syntyi juuri näillä Beckin kuvaamilla tavoilla.

Rajoitteena diplomityöni tekemisessä oli ajan puute. Olen tehnyt diplomityöni työn ohessa ja työn tekemiseen tarvittava aika on pitänyt löytää illoista ja viikonlopuista. Työn puitteissa toteutettua sovellusta olisi voinut tehdä enemmän jos aikaa vaan olisi ollut käytettävissä. Jos sovellusta olisi pystynyt tekemään enemmän niin varmaan arkkitehtuurin kehittymisestäkin olisi löytynyt enemmän analysoitavaa asiaa. Pienen totetutusryhmän voi myös nähdä jonkinlaisena rajoitteena, koska pieni tekijöiden määrä rajoitti jossain määrin sovelluksen toteutuksen laajuutta. Muita suuria rajoitteita diplomityöni tekemisessä ei ollut. Sain määrittää työn suunnan omien näkemyksieni mukaan, alussa määritelyjen raamien puitteissa.

Pieni tiimi toimi ohjelmistoprojektissani erittäin hienosti. Tiimissä oli kaksi ohjelmoijaa, minä ja kollegani, sekä kaksi työn toimeksiantajan edustajaa Europortsilta. Ohjelmointityö sujui kahden henkilön ryhmässä hyvin. Ohjelmointityö oli jaettu niin, että minulla oli vastuullani mobiilisovelluksen ohjelmointi ja kollegani teki web-sovelluksen ohjelmoinnin sekä toteutti tietokantarakenteen. Vaikka molemmilla oli omat osa-alueet ohjelmoinnissa niin yhteistyötä tarvittiin tietysti ohjelmien yhteensovittamiseksi. Myös työn toimeksiantajan kanssa oli helppo toimia. Palaverien ja toteutusjaksojen tavoitteiden sopiminen sujui jouhevasti ja sain palautettakin ihan riittävästi.

Diplomityöni tutkimusmenetelmänä käytettiin toimintatutkimusmenetelmää. Toiminta-tutkimusmenetelmän tavoitteena oli selvittää millainen olisi tehokas ja toimiva mobiiliratkaisu merikonttien kunnon ja laadun valvontaan. Tutkimuksesta saadut tulokset perustuvat sovellusta testanneiden satunnaisten käyttäjien subjektiiviseen näkemykseen. Jos sovellusta olisi testattu kaikkien mahdollisten loppukäyttäjien toimesta niin voi olla, että erilaisia kommentteja ja kehitysehdotuksia olisi tullut enemmän. Esimerkiksi käyttöliittymän käytettävyydestä voisi tulla erilaisia kommentteja, koska sovelluksen käyttökokemus on jokaisella henkilökohtainen.

6. YHTEENVETO

Tässä diplomityössä tutkittiin tietojärjestelmän arkkitehtuurin syntymistä ketterässä ohjelmistokehityksessä. Diplomityön puitteissa toteutettiin merikonttien kunnon ja laadun valvontaan tarkoitettua tietojärjestelmää jolla arkkitehtuurin syntymistä voitiin arvioida. Sovelluksen toteutuksessa keskityttiin toteuttamaan mobiili- ja web-käyttöliittymää sen verran, että työn toimeksiantajan oli mahdollista arvioida olisiko uusi järjestelmä mahdollista toteuttaa kyseisillä välineillä ja teknologioilla.

Tämän diplomityön kolmannessa luvussa käsiteltiin ohjelmistoarkkitehtuuria ketterässä ohjelmistokehityksessä ja kirjoitin Dean Leffingwellin (2007) kirjassa olevasta näkemyksestä ohjelmistoarkkitehtuurin muodostumisesta ketterässä ohjelmistotuotannossa. Leffingwellin kirjassa mainitaan, että ketterässä ohjelmistotuotannossa ei tehdä raskasta suunnitteluprosessia aluksi, vaan järjestelmän rakenne muodostuu asiakkaan ja kehitysryhmän valitsemien toteutettavien ominaisuuksien mukaan. Ohjelmiston arkkitehtuuri muodostuu inkrementaalisesti kehitysjaksosta toiseen. Ohjelmiston arkkitehtuurin inkrementaalinen muodostuminen tarkoitaa sitä, että arkkitehtuuri kasvaa koko ajan kohti lopullista muotoaan.

Toteutetun sovelluksen arkkitehtuurin syntymisen pystyi helposti havaitsemaan ohjelmistoprojektin toteutuksen edetessä. Aluksi toteutettavan sovelluksen arkkitehtuurista oli olemassa jonkinlaista suunnitelmaa mutta lopulliseen muotoonsa sovelluksen arkkitehtuuri kehittyi ohjelmistoprojektin kolmessa toteutusjaksossa. Kuten Leffinwellin kirjassa mainitaan niin myös ketterässä ohjelmistoprojektissani järjestelmän rakenne muodostuu asiakkaan ja kehitysryhmän valitsemien toteutettavien ominaisuuksien mukaan, inkrementaalisesti kehitysjaksosta toiseen.

Arkkitehtuurin kehittymistä voisi jatkossa tutkia useammalla eri tavalla. Jos Container App -sovellusta jatkokehitettäisiin niin mukaan voisi ottaa enemmän ohjelmoijia. Lisäksi työn toimeksiantajan puolelta ohjelmistoprojektiin voisi ottaa mukaan myös kontin tarkastajat. Uusien ohjelmoijien mukaan ottaminen toisi mahdollisuuden kehittää sovellusta nopeammin mutta varmasti alkaisi ilmetä myös jotain haasteita esimerkiksi tehtävien resursoinnin suhteen. Kontin tarkastajat taas voisivat tuoda uutta näkökulmaa sovelluksen kehityksen suuntaan, mutta suurempi joukko henkilöitä työn toimeksiantajan puolella voisi myös vaikeuttaa päätösten tekoa siitä mitä ominaisuuksia kehitysjaksoihin otettaisiin toteutettaviksi.

Jos ohjelmistoprojektiin otettaisiin mukaan useita uusia ohjelmoijia niin silloin voisi harkita ohjelmistoarkkitehtuurin eriyttämistä omaksi prosessikseen kuten Leffingwell (2007) on kirjoittanut. Tässä toimintatavassa erilliset arkkitehdit suunnittelevat ja mahdollisesti myös toteuttavat arkkitehtuurin. Arkkitehtiryhmä keskittyy ohjelmiston

myöhempien kehitysjaksojen vaatimuksiin ja ryhmän vastuulla on suunnitella järjestelmän rakenne niin, että se tukee seuraavia lisättäviä toimintoja.

Container App -sovellus ei tullut diplomityöni puitteissa lähellekään valmiiksi, eikä se ollut tarkoituskaan. Työn toimeksiantaja sai nyt kuitenkin testata ja esitellä Android-laitteelle toteutettua mobiili- ja web-sovellusta mahdollisille loppukäyttäjille sen verran, että näkemys sovelluksen käyttökelpoisuudesta oli mahdollista tehdä.

Sovellusta esiteliin Europortsilla kontin tarkastajille ja yleinen vastaanotto on ollut positiivista. Lopullisena arviona ja näkemyksenä on, että sovelluksessa olisi potentiaalia konttien tarkastusta helpottavaan ja nopeuttavaan ratkaisuun. Itse sovellus vaatisi kuitenkin vielä melko paljon jatkokehittelyä, jotta kaikki tarvittava toiminnallisuus saataisiin siihen toteutettua.

Vaikka perinteistä vesiputousmallia ja ketteriä ohjelmistokehitysmenetelmiä pidetään usein vastakkainasetteluna niin ohjelmistoarkkitehtuuri näyttää vääjäämättä syntyvän molemmissa ohjelmistokehitysmenetelmissä − ohjelmistoarkkitehtuuri syntyy näissä menetelmissä vaan hieman eri tavalla.

LÄHTEET

Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J., 2002. *Agile software development methods, Review and analysis.* [Online]
Available at: http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf
[Haettu 21 08 2017].

Android.com, 2017. *Transferring Data Using Sync Adapters.* [Online]
Available at: https://developer.android.com/training/sync-adapters/index.html
[Haettu 10 10 2017].

Argyris, C., 1982. *Reasoning, Learning, and Action: Individual and Organizational.* San Francisco: Jossey-Bass Publishers.

Babar, M. A., Brown, A. W. & Mistrik, I., 2014. *Agile Software Architecture: Aligning Agile Processes and Software Architectures.* Waltham: Morgan Kaufmann Publishers.

Baskerville, R. L. & Wood-Harper, T., 1998. Diversity in information systems action research methods. *European Journal of Information Systems,* pp. 90-107.

Bass, L., Clements, P. & Kazman, R., 2003. *Software Architecture in Practice (2nd edition).* Boston: Addison-Wesley.

Beck, K., 1999. *Extreme Programming Explained.* Boston: Addison-Wesley Professional.

Beck, K. & Andres, C., 2004. *Extreme Programming Explained: Embrace Change (2nd Edition).* Boston: Addison-Wesley Professional.

Beck, K. ym., 2001. *Ketterän ohjelmistokehityksen julistus.* [Online]
Available at: http://agilemanifesto.org/iso/fi/manifesto.html

Bossavit, L., 2013. *The Leprechauns of Software Engineering.* Victoria, British Columbia: Lean Publishing.

Clements, P., Kazman, R. & Bass, L., 2012. *Software Architecture in Practice, Third Edition.* Boston: Addison-Wesley Professional.

Consultantsea Ltd, 2017. *Budgetshippingcontainers.co.uk.* [Online]
Available at: http://www.budgetshippingcontainers.co.uk/info/how-many-shipping-containers-are-there-in-the-world/
[Haettu 25 10 2017].

Cudahy, B. J., 2006. *Box Boats: How Container Ships Changed the World.* New York: Fordham University Press.

Django Software Foundation, 2017. *Django.* [Online]
Available at: https://www.djangoproject.com/
[Haettu 27 8 2017].

Ebeling, C. E., 2009. "Evolution of a Box". *American heritage of Invention and Technology,* p. 8–9.

Google, 2017. *Android.* [Online]
Available at: https://www.android.com/
[Haettu 27 8 2017].

Haikala, I. & Mikkonen, T., 2011. *Ohjelmistotuotannon käytännöt (12. uud. p. ed.).* Helsinki: Talentum.

Haikala, I. & Märijärvi, J., 2006. *Ohjelmistotuotanto.* Helsinki: Talentum Media Oy.

Hofmeister, C. et al., 2007. A general model of software architecture design derived from five industrial approaches. *The Journal of Systems and Software 80 (2007) 106–126.*

IEEE 2000, 2000. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Standard 1471-2000.* New York: The Institute of Electrical and Electronics Engineers, Inc..

Jansen, A. & Bosch, J., 2005. *Software architecture as a set of architectural design decisions Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA) 2005 IEEE 109–120.* s.l., s.n.

Jönssön, S., 1991. Information Systems Research: Contemporary Approaches & Emergent Traditions, H.-E. Nissen, H.K. Klein and R. Hirschheim, Editors. pp. 371-396.

Koskimies, K. & Mikkonen, T., 2005. *Ohjelmistoarkkitehtuurit.* Tampere: Talentum Oy.

Larman, G., 2004. *Agile and Iterative Development: A Manager's Guide.* Boston: Addison-Wesley.

Leffingwell, D., 2007. *Scaling Software Agility: Best Practices for Large Enterprises.* Boston: Addison-Wesley.

Leffingwell, D. & Widrig, D., 2003. *Managing Software Requirements, Second Edition: A Use Case Approach..* Boston: Addison-Wesley.

McConnell, S., 1996. *Rapid Development: Taming Wild Software Schedules.* Washington: Microsoft Press.

Mistrik, I. ym., 2014. *Relating System Quality and Software Architecture.* Waltham: Morgan Kaufmann Publishers.

Mistrik, I., Bahsoon, R., Kazman, R. & Zhang, Y., 2014. *Economics-Driven Software Architecture.* Waltham: Morgan Kaufmann Publishers.

Nelson, J., 2017. *PostgREST Documentation.* [Online]
Available at: https://postgrest.com/en/v4.3/
[Haettu 10 10 2017].

No-IP.com, 2017. *no-ip.* [Online]
Available at: https://www.noip.com/
[Haettu 27 8 2017].

Python Software Foundation, 2017. *Python.* [Online]
Available at: https://www.python.org/
[Haettu 27 8 2017].

Rapoport, R. N., 1970. Three Dilemmas of Action Research. *Human relations,* Osa/vuosikerta 23, pp. 499-513.

Royce, W. W., 1970. Managing the Development of Large Software Systems. *Proceedings of the 9th International Conference on Software Engineering.*

Schiel, J., 2012. *The ScrumMaster Study Guide.* Boca Raton: Auerbach Publications.

Schmidt, R. F., 2013. *Software Engineering: Architecture-Driven Software Development.* Waltham: Morgan Kaufmann Publishers.

Statista, 2017. *Container Shipping - Statistics & Facts.* [Online]
Available at: https://www.statista.com/topics/1367/container-shipping/
[Haettu 23 10 2017].

Statista, 2017. *Number of merchant ships by type 2016.* [Online]
Available at: https://www.statista.com/statistics/264024/number-of-merchant-ships-worldwide-by-type/
[Haettu 23 10 2017].

Susman, G. I. & Evered, R. D., 1978. An Assessment of the Scientific Merits of Action Research. *Administrative Science Quarterly,* Osa/vuosikerta 23, pp. 582-603.

Thapparambil, P., 2005. Agile architecture: pattern or oxymoron?. *Agile Times 6 2005 ,* p. 43–48.

The PostgreSQL Global Development Group, 2017. *Postgresql.* [Online]
Available at: https://www.postgresql.org/
[Haettu 27 8 2017].

Wallenius, J., 2016. Kova paketti muutti maailman. *Turun Sanomat, 24.4.2016, 112. vsk, nro 112, s. 14. Turku: Turun Sanomat Oy. ISSN 0356-133X..*

Wells, D., 1999. *The Rules of Extreme Programming.* [Online]
Available at: http://www.extremeprogramming.org/rules.html
[Haettu 30 08 2017].

VersionOne, 2017. *The 11th annual State of Agile Report.* [Online]
Available at: https://explore.versionone.com/state-of-agile/versionone-11th-annual-state-of-agile-report-2
[Haettu 27 8 2017].

World Shipping Council, 2017. *Top 50 World Container Ports.* [Online]
Available at: http://www.worldshipping.org/about-the-industry/global-trade/top-50-world-container-ports
[Haettu 23 10 2017].